Optimizing Analytical Queries over Semantic Web Sources

Ibragimov, Dilshod

DOI (link to publication from Publisher):
10.5278/vbn.phd.tech.00022

Publication date:
2017

Document Version
Publisher's PDF, also known as Version of record

OPTIMIZING ANALYTICAL QUERIES OVER SEMANTIC WEB SOURCES

BY
DILSHOD IBRAGIMOV

DISSERTATION SUBMITTED 2017

AALBORG UNIVERSITY
DENMARK

Optimizing Analytical Queries over Semantic Web Sources

Ph.D. Dissertation
Dilshod Ibragimov

Dissertation submitted September 2017

A thesis submitted to the Technical Faculty of IT and Design at Aalborg University (AAU) and the Faculty of Engineering at Université Libre de Bruxelles (ULB), in partial fulfillment of the requirements within the scope of the IT4BI-DC programme for the joint Ph.D. degree in computer science. The thesis is not submitted to any other organization at the same time.

Dissertation submitted:	September 2017
ULB PhD Supervisor:	Prof. Esteban Zimányi Université Libre de Bruxelles, Belgium
PhD supervisor:	Prof. Torben Bach Pedersen Aalborg University, Denmark
Assistant PhD supervisor:	Assoc. Prof. Katja Hose Aalborg University, Denmark
ULB PhD Committee:	Assoc. Prof. Stijn Vansummeren Université Libre de Bruxelles, Belgium
	Prof. Toon Calders University of Antwerp, Belgium
	Prof. Maria-Esther Vidal Universidad Simón Bolívar, Venezuela
	Prof. Ladjel Bellatreche National Engineering School for Mechanics and Aerotechnics (ENSMA), France
PhD committee:	Associate Professor Kristian Torp (chairman) Aalborg University, Denmark
	Professor Maria-Esther Vidal Universidad Simón Bolívar, Venezuela
	Professor Ladjel Bellatreche National Engineering School for Mechanics and Aerotechnics (ENSMA), France
PhD Series:	Technical Faculty of IT and Design, Aalborg University
Department:	Department of Computer Science

ISSN (online): 2446-1628
ISBN (online): 978-87-7210-072-2

Published by:
Aalborg University Press
Skjernvej 4A, 2nd floor
DK – 9220 Aalborg Ø
Phone: +45 99407140
aauf@forlag.aau.dk
forlag.aau.dk

Printed in Denmark by Rosendahls, 2017

Abstract

Data has always been a key asset for a variety of industries and businesses but lately it is giving data owners a true competitive advantage over others. Nowadays companies collect big volumes of data and store them in large multidimensional databases called data warehouses. A data warehouse presents aggregated data as a cube where cells of the cube contain facts and contextual information such as dates, locations, customer and supplier info, etc. Data warehouse solutions successfully employ Online Analytical Processing (OLAP) to analyze these large sets of data, e.g., sales data can be aggregated along the location and/or time dimension. Currently, new challenges are set by recent trends in technology and the Web. Much information is available on the Web in a machine-processable form (Semantic Web) and Business Intelligence (BI) tools need to be able to discover and retrieve relevant information and present it to users to aid in proper analysis of the situation. Many government and other organizations make their data openly available, identify their data with Uniform Resource Identifiers (URI), and interlink data to other data. This collection of interrelated datasets on the Web is called Linked Data [1]. These datasets are based on the Resource Description Framework (RDF) – a standard format for data interchange on the Web [2]. SPARQL, a query language and protocol for RDF [3], is used to query and manipulate RDF datasets stored in SPARQL endpoints. SPARQL 1.1 Federated Query [4] also defines an extension for executing queries distributed over several SPARQL endpoints. Thus, current standards enable complex analytical queries over multiple data sources and integrating these data into the analysis process becomes a necessity for BI tools. However, due to the amount and complexity of data available on the Web, incorporation and utilization of these data are not easy and straightforward. Therefore, an efficient OLAP solution over Semantic Web sources is needed to improve BI tools.

This PhD thesis focuses on the challenges related to the optimization of analytical queries that retrieve data from multiple SPARQL endpoints. First, the thesis proposes a framework for the discovery, integration, and analytical querying of Linked Data – this type of OLAP was termed *Exploratory*

OLAP [5]. The framework is designed to use a multidimensional schema of the OLAP cube expressed in RDF vocabularies to be able to query data sources, extract and aggregate data, and build a data cube. We also propose a computer-aided process for discovering previously unknown data sources and building a multidimensional schema of the cube. Second, due to the inefficient execution of analytical federated SPARQL queries by state-of-the-art SPARQL endpoints, this thesis proposes a set of query processing strategies and the associated Cost-based Optimizer for Distributed Aggregate queries (CoDA) for optimizing the execution of such analytical SPARQL queries. Third, to overcome the challenges for aggregate SPARQL query processing techniques on a single endpoint, we propose MARVEL (MAterialized Rdf Views with Entailment and incompLetness) – an approach that uses RDF specific materialized view techniques to process complex aggregate queries. The approach consists of a view selection algorithm based on an associated RDF-specific cost model, a view definition syntax, and an algorithm for rewriting SPARQL queries using materialized RDF views. Lastly, we focus on techniques to support analytical SPARQL queries over related data located at multiple endpoints which enable interesting insights and analyses at a large scale. In particular, the proposed technique is able to integrate the diverse schemas of SPARQL endpoints and provide access to the data via OLAP-style hierarchies to enable uniform, efficient, and powerful analytics. Finally, the developed techniques advocate for a greater attention to analytical query processing in distributed RDF data systems.

Resumé

Data har altid været essentielt for en bred vifte af virksomheder, endvidere giver det i højere og højere grad virksomheder en målbar fordel over deres konkurrenter. Moderne virksomheder indsamler og opbevarer deres data i multidimensionelle databaser kaldet datavarehuse. Et datavarehus repræsenterer dets data som en kube, hvor cellerne indeholder fakta og kontekstuel information såsom tid, lokation, kunder, leverandører etc., som dimensioner. Datavarehuse bruger Online Analytisk Processering (OLAP) til at analysere store datamængder. For eksempel salgsdata kan blive aggregeret over lokation- og/eller tids -dimensionerne. Nye teknologiske trends skaber nye udfordringer. Mere og mere data er tilgængeligt på internettet i maskinforståeligt format (Semantisk Web). Business Intelligence (BI) værktøjer er dog endnu ikke i stand til at indsamle og præsentere dette data til brugere så det kan blive analyseret. Flere og flere organisationer har gjort deres data frit tilgængelige på internettet, heriblandt mange offentlige institutioner, ved blandt andet at gøre deres data identificerbart via Uniform Resource Identifiers (URIs) og skabe referencer til andre relevante datasamlinger. Dette sæt af datasamlinger der er bundet sammen med referencer hedder Linked Data [1]. Disse datasamlinger er alle baseret på Resource Description Framework (RDF) - et standard format til udveksling af data på internettet [2]. SPARQL, et forespørgsels-sprog og -protokol til RDF [3], bruges til at lave forespørgsler og manipulere RDF datasamlinger som er lagret i SPARQL endepunkter. SPARQL 1.1 fødererede forespørgsler [4] er en udvidelse der gør det muligt at eksekvere forespørgsler over flere SPARQL endepunkter. Det betyder at aktuelle standarder gør det muligt at foretage komplekse analytiske forespørgsler over flere datakilder og dermed integrere disse datasamlinger i BI værktøjer. Dog, på grund af mængden af data og dets kompleksitet er inkorporering og brug hverken nemt eller ligetil. Derfor er effektive OLAP løsninger til det Semantiske Web nødvendigt for at forbedre de eksisterende BI værktøjer.

Denne PhD afhandling fokuserer på udfordringerne relateret til optimering af analytiske forespørgsler over flere SPARQL endepunkter. For det første foreslår denne afhandling et system til opdagelse, integration og ek-

sekvering af analytiske forespørgsler over Linked Data - denne type af OLAP kaldes *Exploratory* OLAP [5]. Systemet er designet til at bruge et multidimensionelt skema for en OLAP kube, udtrykt med RDF vokabularier, til eksekvering af forespørgsler over datakilder, udtrække og aggregere data, og konstruere en datakube. Endvidere, foreslår vi en computerstøttet proces til at opdage ukendte datakilder og konstruktion af det tilsvarende multidimensionelle skema. For det andet, på grund af ineffektiv eksekvering af analytiske fødererede SPARQL forespørgsler over førende SPARQL endepunkter, foreslår denne afhandling, et sæt af forespørgselsprocesseringsstrategier og de tilhørende pris-baserede optimeringer for distribuerede aggregeringsforespørgsler (CoDA) for optimering af sådanne forespørgsler. For det tredje, for at overkomme udfordringerne associeret med processeringsteknikker for SPARQL aggregeringsforespørgsler over et enkelt endepunkt, foreslår vi MARVEL (MAterialized Rdf Views with Entailment and incompLetness), som bruger RDF-specifikke view materialiseringsteknikker til at processere komplekse aggregeringsforespørgsler. Dette system består af en view-udvælgelses algoritme baseret på en RDF-specifik pris-baseret model, en syntaks for definition af views og en algoritme til at omskrive SPARQL forespørgsler til at bruge de materialiserede views. For det fjerde, fokuserer vi på teknikker til at understøtte analytiske SPARQL forespørgsler over data der eksisterer spredt over flere forskellige endepunkter. Dette gør det muligt at skabe forståelse og analyse af store datasamlinger. Den foreslåede teknik gør det muligt at integrere forskellige skemaer, dette gør det muligt at tilgå data samlingerne via et OLAP-lignende hierarki, og skaber mulighed for at foretage ensartede, effektive og vigtige analyser. Overordnet skaber de udviklede teknikker større mulighed for analytiske forespørgsler over distribuerede RDF datasystemer.

Résumé

Les données ont toujours été un atout clé pour beaucoup d'industries et d'entreprises ; cependant, ces derniers temps les possesseurs de données jouissent d'un véritable avantage compétitif sur les autres. De nos jours, les compagnies collectent de gros volumes de données et les stockent dans de grandes bases de données multidimensionnelles appelées entrepôts de données. Un entrepôt de données présente les données agrégées sous la forme d'un cube dont les cellules contiennent des faits et des informations contextuelles telles que des dates, des lieux, des informations sur les clients et fournisseurs, etc. Les solutions d'entreposage de données utilisent avec succès OLAP (Traitement Analytique En Ligne – en anglais Online Analytical Processing) afin d'analyser ces grands ensembles de données ; par exemple, les informations des ventes peuvent être agrégées selon le lieu et/ou la dimension temporelle. Les tendances récentes des technologies et du Web posent actuellement de nouveaux défis. Une bonne quantité de l'information disponible sur le Web s'y trouve sous une forme qui se prête au traitement par machine (Web Sémantique) ; les outils de veille économique (en anglais Business Intelligence ou BI) doivent être capables de découvrir et récupérer les informations pertinentes, et les présenter aux utilisateurs afin de les assister dans une bonne analyse de la situation. De nombreux gouvernements et organisations rendent leurs données publiquement accessible, identifiables avec des URI (Unified Resource Identifiers), et les lient à d'autres données. Cette collection de jeux de données interconnectés sur le Web s'appelle Linked Data [1]. Ces jeux de données sont basés sur le modèle RDF (Resource Description Framework) – un format standard pour l'échange de données sur le Web [2]. SPARQL, un protocole et un langage de requêtes pour RDF [4], est utilisé pour interroger et manipuler les jeux de données RDF stockés dans des triplestores SPARQL. SPARQL 1.1 Federated Query [6] définit également une extension pour exécuter des requêtes distribuées sur plusieurs triplestores. Le standard actuel permet donc des requêtes analytiques complexes sur de multiples sources de données, et l'intégration de ces données dans le processus d'analyse devient une nécessité pour les outils de BI. Cependant, en raison de la quantité et de la complexité des données

disponibles sur le Web, leur incorporation et leur utilisation ne sont pas toujours évidentes. Par conséquent, une solution OLAP efficace sur des source Web Sémantiques est nécessaire pour améliorer les outils de BI.

Cette thèse de doctorat se concentre sur les défis liés à l'optimisation des requêtes analytiques qui utilisent des données provenant de plusieurs triplestores SPARQL. Premièrement, cette thèse propose un framework pour la découverte, l'intégration et l'interrogation analytique des Linked Data – ce type d'OLAP a été nommé OLAP Exploratoire [21]. Ce framework est conçu pour utiliser un schéma multidimensionnel du cube OLAP exprimé dans des vocabulaires RDF, afin de pouvoir interroger des sources de données, extraire et agréger des données, et construire un cube de données. Nous proposons également un processus assisté par ordinateur pour découvrir des sources de données précédemment inconnues et construire un schéma multidimensionnel du cube. Deuxièmement, vu l'inefficacité actuelle des triplestores SPARQL pour l'exécution des requêtes analytiques fédérées, cette thèse propose un ensemble de stratégies pour le traitement de ces requêtes ainsi qu'un module (appelé Cost-based Optimizer for Distributed Aggregate ou CoDA) pour optimiser leur exécution. Troisièmement, afin de surmonter les défis liés aux techniques de traitement des requêtes SPARQL agrégées sur un seul triplestore, nous proposons MARVEL (MAterialized Rdf Views with Entailment and incompLeteness) – une approche qui utilise des techniques de vues matérialisées spécifiques à RDF pour traiter les requêtes agrégées complexes. Notre approche consiste en un algorithme de sélection de vues selon un modèle de coût associé spécifique à RDF, une syntaxe pour la définition des vues et un algorithme pour la réécriture des requêtes SPARQL en utilisant les vues matérialisées RDF. Finalement, nous nous concentrons sur les techniques relatives au support des requêtes analytiques SPARQL sur des données liées situées en de multiples triplestores, qui nous conduisent à d'intéressantes analyses et constatations à grande échelle. En particulier, la technique proposée est capable d'intégrer les schémas divers des endpoints SPARQL, donnant accès aux données via des hiérarchies dans le style d'OLAP pour permettre des analyses uniformes, efficaces et puissantes. Enfin, cette thèse préconise une plus grande attention au traitement des requêtes analytiques au sein des systèmes RDF distribués.

Contents

Chapter 1

Introduction

1 Semantic Web

Designed initially for humans, the Web has gone through several stages of development. Nowadays, the Web is considered to be a space for publishing the information so that *both* humans and machines can effectively retrieve and process it. The World Wide Web Consortium (W3C) came up with the standards to promote common data formats and exchange protocols on the Web to enable machine-to-machine communication. This set of standards is gathered under the common umbrella term Semantic Web.

In the Semantic Web, the information is given a well-defined meaning and machines are able to understand the information and perform sophisticated tasks on behalf of the users. The meaning is expressed by Resource Description Framework (RDF) [2], which encodes it as a set of triples. An RDF triple contains three components: the subject (a URI reference or a blank node), the predicate, (a URI reference) and the object (a URI reference, a literal or a blank node). Such a structure allows to assert that a particular concept (subject) has a property (predicate) with the certain value (object) and allows to describe the vast majority of the data processed by machines.

A triple can be modeled as a directed link from the subject to the object labeled by the predicate. The interrelated set of triples, thus, can be represented as a graph where vertices represent subjects or objects and the predicates are the edges connecting the vertices. This data model has a semi-structured nature – it is heterogeneous (resources may have different properties) and self-describing (the structure is encoded in the data itself). These characteristics make RDF a suitable format for the data interchange on the Web.

An integral part of RDF – RDF Schema (RDFS) [6] – provides a data-modeling vocabulary for RDF data. It defines a mechanism for describing groups of associated resources and relationships between these resources.

The classes and properties constitute the basic constructs of RDFS. Classes group related resources. Properties model a relation between subject resources and object resources. For example, *rdfs:subClassOf* models a subclass relationship and *rdfs:subPropertyOf* models a sub-property relationship for classes and properties. The classes of the subjects and objects of the triples with the predicate p can be defined by *rdfs:domain* and *rdfs:range* properties of that predicate. *rdfs:range* identifies the class of the triples' objects while *rdfs:domain* identifies the class of the triples' subjects.

Besides an abstract syntax that defines a data model, RDF specification also defines semantics which provides a basis for reasoning about the meaning of RDF statements. In particular, the RDF specification supports the notions of entailment to provide a basis for defining inference rules in RDF data. Based on the RDF data and the inference rules, implicit RDF triples can be derived.

There are different types of entailment rules in RDF. The first type of rules generalizes triples using blank nodes (blank nodes *:b* are variables, which can be used as subjects and/or objects of RDF statements). For example, RDF triple `student hasSupervisor supervisor` entails another RDF triple `student hasSupervisor :b`. Another type of rules infers triples from the semantics of built-in classes and properties. For instance, if `s rdf:type o` then o is a class: `o rdf:type rdfs:Class`. Finally, the third type of rules derives inferred triples from RDFS constraints. The rules derive inferred triples through the transitivity of class and property inclusions and from inheritance of domain and range types. Table 1.1 provides some examples of the entailed triples.

RDFS Properties	Explicit Triple	Implicit Triple
(c_1, rdfs:subClassOf, c_2)	(s, rdf:type, c_1)	(s, rdf:type, c_2)
(p_1, rdfs:subPropertyOf, p_2)	(s, p_1, o)	(s, p_2, o)
(p, rdfs:range, c)	(s, p, o)	(o, rdf:type, c)
(p, rdfs:domain, c)	(s, p, o)	(s, rdf:type, c)

Table 1.1: RDFS Entailment Rules

For the data represented using the RDF framework, W3C proposes to query these data using the SPARQL [3] language. The SPARQL language is used for RDF graph pattern matching to extract tabular information from a graph or to construct new RDF graphs. SPARQL has 4 query forms which use the solutions from pattern matching to represent result sets or RDF graphs. They are *SELECT*, *CONSTRUCT*, *ASK* and *DESCRIBE*. The *SELECT* query form returns a subset of the variables bound in a query pattern match. The *CONSTRUCT* query form returns a new RDF graph constructed using the triple patterns and variables from a graph pattern. The *ASK* query form

returns a boolean indicating whether a specific query pattern match is found in the graph or not. The *DESCRIBE* query form returns an RDF graph that describes the specific resource.

The latest version of SPARQL 1.1 [7] introduced a new set of features in comparison with SPARQL 1.0 like the aggregate functions, subqueries, negation, update queries and federated queries. Aggregates, for example, calculate values over groups of solutions. Grouping is defined by the *GROUP BY* syntax and is defaulted to a single group containing all solutions on the absence of the *GROUP BY* statement. Version SPARQL 1.1 defines *COUNT*, *SUM*, *MIN*, *MAX*, *AVG*, *GROUP_CONCAT*, and *SAMPLE* aggregates.

SPARQL 1.1 Federated Query W3C Recommendation [4] defines the syntax and semantics of SPARQL 1.1 Federated Query extension for executing queries distributed over several SPARQL endpoints. The queries are extended using the *SERVICE* keyword in SPARQL 1.1 which directs a portion of the query to another SPARQL endpoint. The results are then combined with the results from the other portion of the query.

To enable the exploration of the data by persons and/or machines, Tim Berners-Lee suggested to *link* data resources on the Web using RDF [8]. He suggested the following principles for Linked Data:

1. URIs should be used as object identifiers
2. HTTP URIs should be used to look up referenced objects
3. When someone looks up a URI, useful information should be provided using standards (RDF, SPARQL)
4. Semantic links to other URIs should lead to the discovery of more information

These 4 principles became recommended best practices for exposing, sharing, and connecting data, information, and knowledge on the Semantic Web. To facilitate even greater use of the Linked Data on the Web, it is suggested to release the data under an open license. Such data is called Linked Open Data (LOD).

Figure 1.1 represents the latest state of the LOD cloud. This figure depicts all known RDF datasets that have been published in Linked Data format. As can be seen, the datasets are highly interconnected. Thus, it is possible to discover new data or analyze data from different perspective by using the links between the datasets.

With more and more data available as LOD, companies see the benefit of integrating these data with the data from their proprietary systems or querying these data to retrieve the information useful in the decision making process. Typically, the solutions involving the use of external data

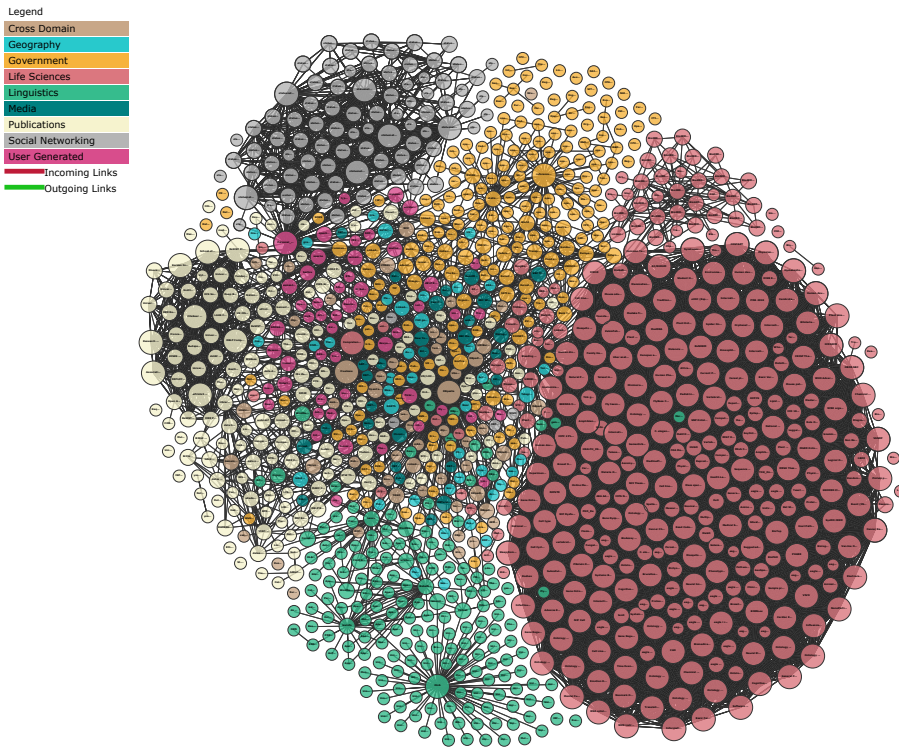


Fig. 1.1: Linking Open Data cloud diagram 2017, by Andrejs Abele, John P. McCrae, Paul Buitelaar, Anja Jentzsch and Richard Cyganiak. http://lod-cloud.net/

employed semi-automatic methods for on-demand extracting and combining large amounts of semantic data expressed in RDF into multidimensional structures suitable for BI analysis. The common architecture of such BI systems contains an ETL (Extract, Transform and Load) pipeline that extracts, transforms, and loads data from the external sources into a relational/multidimensional database. Then, BI systems use OLAP tools in these databases to analyze the loaded data. However, such an approach has some drawbacks.

First of all, changes in the external data sources may lead to changes in the structure of a relational database (changes in the schema) and will impact the entire Extract-Transform-Load process to have the changes propagated. Second, RDF systems support triples with blank nodes (triples with unknown components) whereas relational systems require that all attributes either have some value or null. Third, in comparison to relational systems, native RDF systems are better at handling the graph-structured RDF model and other

RDF specifics like entailment, when new information can be derived from the data using RDF semantics (standard relational databases are limited to explicit data only). Thus, using relational databases that have been developed over the decades for manipulating plain, structural data is inappropriate for storing and processing massive unstructured and *semantic rich* datasets.

Given the complexity of information needs for BI applications, some queries can only be answered by retrieving the results stored across different data sources. With the recent SPARQL 1.1 standard, users may formulate a single query that accesses data from federated web sources to conduct complex analysis tasks involving grouping, aggregation and retrieval of data from multiple sources. However, as both aggregate and federated queries have become available only recently, the performance of state-of-the-art triple stores during the execution of such queries is unacceptable – the queries often time out or take unjustifiably long to execute. Thus, the optimization of engines that support these types of queries are desired to ensure the usability of LOD in BI scenarios. Therefore, in this thesis, we investigate RDF data analytics over federated data sources in a context of BI applications considering the features that have given RDF its popularity, namely heterogeneity, rich semantics, entailment, ease of publication, etc.

2 Thesis Overview

This section gives an overview of each chapter in this thesis and outline the overall contributions. An overview of how the chapters build on and relate to each other is illustrated in Figure 1.2. Chapter 1 introduces the thesis. Chapters 2 to 5 present the main content and contributions of the thesis. Chapter 2 proposes a framework for *Exploratory* OLAP over RDF data sources and presents a use case to demonstrate its applicability. The chapter lays the foundation for further investigation of various aspects of the framework, thus Chapters 3 and 5 originate from Chapter 2. In Chapter 3, we consider issues faced by state-of-the-art systems during the execution of aggregate queries in a federation of SPARQL endpoints, which often lead to timeouts in query execution, and propose optimizations needed for the efficient processing of these queries. We identify three strategies suitable for evaluating aggregate queries in a federated setup and propose a cost model to choose between them. However, in Chapter 3, we do not consider federations in which multiple *related* data sources (with similar data models) are virtually integrated into a common single logical data source. Thus, in Chapter 4, we address this issue and propose a solution that enables the analysis of data *across* multiple endpoints. Chapter 5 considers an issue of performance optimization for aggregate SPARQL queries on a single endpoint, unlike Chapters 3 and 4, which analyze the issue in a federated setup. Performance optimization on

a single endpoint contributes to the overall efficiency of the framework proposed in Chapter 2 thus we designate a separate chapter of the thesis to this problem. In Chapter 6, we conclude our investigation for all chapters and define the outlook for future research. The following subsections provide more details about Chapters 2 to 5.

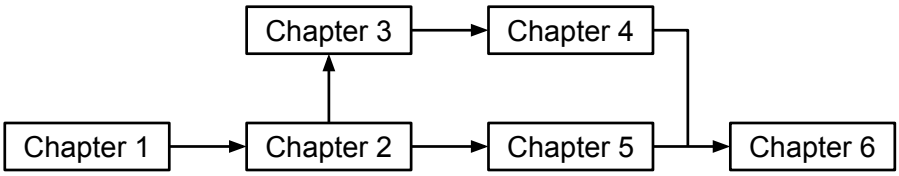


Fig. 1.2: Thesis content overview

2.1 Chapter 2: Towards Exploratory OLAP over Linked Open Data – A Case Study

For many years, BI tools provide fundamental support in analyzing big volumes of information for intelligent decision making. Traditionally, companies use large multidimensional databases called data warehouses (DW) to store big volumes of information and Online Analytical Processing tools to analyze it.

As more and more data sources become available on the Web in form of LOD, companies see the opportunity to take this information into account. BI tools should discover and retrieve relevant information and present it to users to aid in proper analysis of the situation. However, due to the amount and complexity of data available on the Web, incorporation and utilization of these data are not easy and straightforward.

In Chapter 2, we introduced the conceptual model of a system for *Exploratory* OLAP over RDF data sources. The system builds the required multidimensional schema of the OLAP cube using such RDF vocabularies as QB4OLAP [9] and VoID [10]. QB4OLAP is an RDF vocabulary that allows the publication of multidimensional data. QB4OLAP can represent dimension levels, level members, roll-up relations between levels and level members, aggregate functions applied to the measures, etc. Thus, we use QB4OLAP for building data cube schemas. VoID is an RDF Schema vocabulary for expressing metadata about RDF datasets. The vocabulary may specify how RDF data can be accessed using various protocols. Combining these two vocabularies allows the system to identify the endpoints that need to be queried and build semantic queries routed to necessary data sources.

We have identified four modules needed in such systems. The Global

Conceptual Schema module contains information about the schema of the specified data cube expressed in QB4OLAP and VoID vocabularies. It is responsible for storing appropriate information and providing this information to other modules of the system. The Semantic Query Processor is a module of the system that accepts a user query as input and produces a multidimensional SPARQL query using Global Conceptual Schema module for further processing. The Source Discovery/Schema Builder module is responsible for deriving a schema of the OLAP cube based on user requirements. This module interacts with the user during the schema construction phase. Finally, the Distributed Query Processing module is responsible for executing queries over different data sources, merging and computing the final results. Following the common approach in distributed database systems [11], we propose that the Distributed Query Processing module contains a mediator/wrapper for splitting the user query and executing queries over various data sources. The results received from the wrappers will then be merged by the mediator and passed to the user.

Discovering previously unknown data sources for the given data cube is an important part of Exploratory OLAP systems. In our framework, we propose to have a dedicated module for the Source Discovery/Schema Building. We elaborated on existing approaches for RDF source discovery [12–16] and extended it. We identified three potentially interesting data source discovery approaches for further investigation. The first approach is querying large knowledge bases such as DBpedia, Yago, or Freebase to find relevant information. Querying knowledge bases for the term of interest may lead to the discovery of useful sources of data or the necessary information itself. The second approach for source discovery is querying data management platforms like Datahub which provide an API for searching the data for external applications. The third approach we considered is querying semantic web search engines such as Sindice, which also provides an API for searching data sources.

In this chapter, we presented a use case to demonstrate the applicability of the proposed framework. We defined future research directions and laid the foundation for further investigation of various aspects aiming at the development of a framework for an efficient Exploratory OLAP solution. In the next chapters, we consider optimization techniques used to optimize the performance of aggregate SPARQL queries in a federated setup and on standalone endpoints.

2.2 Chapter 3: Processing Aggregate Queries in a Federation of SPARQL Endpoints

SPARQL 1.1 allows users to formulate a single complex query that involves grouping, aggregation, and retrieval of data from multiple SPARQL end-

points. However, state-of-the-art federation systems lack sophisticated optimization techniques that facilitate efficient execution of such queries over large datasets which often ends up with a timeout of a query. Thus, Chapter 3 discusses the challenges and explores optimizations needed for the efficient processing of aggregate queries in a federation of SPARQL endpoints.

We identify three strategies suitable for evaluating aggregate queries in a federated setup along with the cost model to choose between them. The first strategy is based on the mediator join technique, where the query optimizer at the mediator splits queries into subqueries and sends these subqueries to the target endpoints. The mediator then combines the results received from the endpoints, groups and aggregates them, and displays the final results to the user. The second strategy is based on the bound join or semi-join [17, 18] technique using *UNION*, *FILTER* or *VALUES* constructs. The main principle of this strategy is to execute the mutually disjoint subqueries with the smallest results first and use the retrieved results as bindings for the join variables in other subqueries. The third strategy can be used if an original query can be decomposed to several subqueries such that partial grouping and aggregation can be applied to one of these subqueries. The goal is to reduce the size of the partial results and, consequently, the size of the results for other queries used in mediator join technique.

To identify the best executing strategy for a given query, the system needs to estimate the execution cost of all strategies. For this purpose, we use CoDA (Cost-based Optimizer for Distributed Aggregate Queries) – a cost-based optimizer that finds the best execution plan by computing query execution costs for several alternatives. The cost in CoDA consists of two components – communication and processing costs. The cost components are based on cost factors and result size estimations retrieved using different statistics. The cost factors are continuously calibrated using updated statistics and probing queries.

The comprehensive experiments show that CoDA significantly improves performance over current state-of-the-art systems. The cost-based optimizer consistently picks the optimal strategy which results in the successful execution of the queries over all tested dataset sizes, while other federated systems often timed out at the same settings. The benefit of the system appears clearer with the increase in the number of endpoints in the federation – the CoDA executed all queries where the state-of-the-art systems failed even for elementary queries.

Chapter 3 initiated our research on SPARQL query processing in a federated setup. We discussed the challenges and explored optimizations needed for the efficient processing of aggregate queries in a federation of SPARQL endpoints. However, in this chapter, we did not consider federations in which multiple *related* data sources (with similar data models) are virtually integrated into a common single logical data source to allow for the *integrated*

reporting of the data from those multiple sources. In relational terms, this setup is called a horizontal federation. Thus, we continued our investigation of analytical SPARQL query processing in a federated setup in Chapter 4 by addressing the issue of supporting analytical SPARQL queries over virtually integrated multiple data sources with similar data models.

2.3 Chapter 4: Efficient Support of Analytical SPARQL Queries in Federated Systems

Public data is considered to be one of the resources to put economies onto a high and sustainable growth path. International institutions such as UN, EU, the World Bank as well as governments of many countries openly publish data related to geographical information, statistics, weather data, data from publicly funded research projects, etc. Thus, we can often find *related* data at multiple endpoints. Therefore, in Chapter 4, we consider federated systems where multiple endpoints contribute data for a common "aspect" (dimensions, hierarchies, facts, etc.) and propose LITE (O**L**AP-style Analyt**I**cs in a Federa**T**ion of SPARQL **E**ndpoints) – a system for computing aggregate SPARQL queries over a federation of SPARQL endpoints. As a typical scenario, consider a setup when each endpoint in a federation contains statistics or census data of a single country. In this case, LITE enables the analysis of data *across* these endpoints (i.e. countries) using new dimensions or hierarchies like aggregating data at the continent level. In particular, LITE is able to integrate the diverse schemas of SPARQL endpoints and provide access to the data via OLAP-style hierarchies to enable uniform, efficient, and powerful analytics.

LITE is designed to provide an efficient support of analytical queries over federations of SPARQL endpoints. It is a native RDF/SPARQL-based approach that uses a mediated (global) schema comprising (the relevant parts of) all heterogeneous source schemas (local schemas). A user of LITE does no longer have to be aware of the underlying federation of SPARQL endpoints but can conveniently formulate a query on the global schema and the system will automatically take care of all actions that are necessary to retrieve the final result. For this purpose, LITE models source and target schemas as RDF schema graphs that *highlights* the structure of the data available for the analysis. It then divides local and global schemas to meaningful fragments – the subgraphs that reflect same core concepts in schemas, like a hierarchy step – and maps these fragments to each other. We extend the SPARQL Inferencing Notation (SPIN) Syntax vocabulary [20] to map schema fragments and link to each other nodes in global and local schemas and introduce several property names for this purpose. The extended SPIN vocabulary not only allows us to encode and map graph patterns of arbitrary complexity, but also to represent fragments that reside on a remote endpoints to truly enable a federated

setup.

With schema mappings specified, LITE can rewrite user queries defined over the global schema to corresponding queries over local datasets. The query rewriting algorithm identifies the global schema subgraph that corresponds to the user query by matching the graph patterns of the user query against the RDF schema graph. Then, it identifies the subset of the global schema fragments that constitute the previously identified subgraph. Using these schema fragments and the defined mapping among the schema fragments, LITE builds a subgraph of the local schema that corresponds to the global query for every related data source. Next, the algorithm generates the new query over the local data source based on global query and local schema subgraph. Afterward, LITE applies heuristics to optimize rewritten queries. The aggregate functions in the query are rewritten accordingly. For instance, a non-distributive function AVG will be rewritten using SUM and COUNT. During query rewriting, LITE also applies RDFS rules to account for implicitly defined hierarchies in local datasets (those identified with *rdfs:subClassOf* and *rdfs:subPropertyOf* predicates). Finally, these queries are executed and the results from remote endpoints are merged by the mediator node.

The experimental evaluation of LITE show that it can significantly optimize analytical queries in a federated setup. We observed that with the increase of the nodes in a federated system, LITE is capable of showing stable and reliable performance while state-of-the-art systems fail in the same settings. The advantage of LITE is even more evident for queries that retrieve hierarchical data from remote endpoints. In comparison with the tested system, LITE was up to 7x times faster and scaled well when the data volumes and number of endpoints grew.

In Chapter 4, we continued our research started in Chapter 3. We designed a system to enable the analysis of *related* data *across* different SPARQL endpoints by building a virtually integrated schema over heterogeneous source schemas and rewriting a user query to corresponding local queries. Our approach also takes into the account RDF specifics. However, when experimenting with analytical queries, we noticed that the endpoints process and aggregate large volumes of data which leads to high response times. Thus we decided to address the issue of performance optimization on single endpoints to decrease the overall execution time for aggregate SPARQL queries in a federated setup. We considered the optimization of aggregate SPARQL query execution using materialized views in the next chapter.

2.4 Chapter 5: Optimizing Aggregate SPARQL Queries Using Materialized RDF Views

Naturally, SPARQL endpoints are publicly available and allow the execution of the queries of any complexity. Therefore, the endpoints on the Web may be

exposed to heavy workloads. Such workloads may affect the performance of SPARQL endpoints and be the cause of their low availability [19]. To address this problem, materialized views can be created and used as a source of precomputed partial results during query processing, thus helping to answer queries using less computational resources.

Chapter 5 presents MARVEL (MAterialized Rdf Views with Entailment and incompLetness) – a materialized view selection and analytical SPARQL query rewriting approach for RDF data. Unlike materialized view techniques proposed for relational databases, MARVEL supports RDF specifics, such as incompleteness and the need to support implicit (derived) information. The approach consists of a view selection algorithm based on an associated RDF-specific cost model, a view definition syntax, and an algorithm for rewriting SPARQL queries using materialized RDF views.

Dependencies between all possible views are represented using a data cube lattice – a schema with connected nodes, where a node represents an aggregation by a given combination of dimensions. The lattice formalizes which views (nodes) can be used to evaluate a user query – given a query grouping (*GROUP BY*), the lattice node with the exact same grouping (and its ancestors) can be used. We use the number of triples contained in the materialized view used to answer the query as its cost. However, in the cost model, we also account for RDF specifics, such as incomplete views and complex and indirect hierarchies. We use QB4OLAP to describe the schema of the dataset and extend this schema with information about the completeness of levels, the patterns for defining hierarchy steps, the types of hierarchy levels, etc. Before selecting the views to materialize we account for derived triples (based on existing data and specified semantics only) as these triples constitute the part of the graph. Our algorithm selects N views with the maximum benefit for materialization.

Since RDF views need to be stored as RDF triples, the new triples that constitute views must be created based on the aggregated information. View defining queries for aggregate views are more complex than views for conjunctive queries due to the need to group and aggregate the original data. Additionally, aggregate queries return data in a tabular format, not triples. Thus, the views need to define a new graph structure using the *CONSTRUCT* clause. The subjects of new triples are created by combining the values for variables in the *GROUP BY* clause of a *SELECT* query since the combination of these values is unique.

After materialization, the views can be used to answer user queries. Our query rewriting approach consists of two algorithms: for identifying the best view and for rewriting the query using the selected view. To rewrite a user query, we compare the hierarchy levels of views and the user query and identify the views where the hierarchy levels of all dimensions defined in the view do not exceed the needed hierarchy levels of the query and that the

set of aggregate expressions defined in a view can be used to compute the aggregations defined in the query. Among appropriate views we select the view with the minimum cost. Then, we rewrite a user query by replacing the common roll-up path of the selected view and the user query by the triple patterns from the CONSTRUCT clause of the view. Our algorithm also compares the aggregate functions of the query and the view and identifies those that are needed for rewriting. We also account for the type of the function – algebraic or distributive. The triple patterns of the view are placed inside the GRAPH statement of the rewritten query to account for the different storage of the view triples.

The experimental results showed the advantage of using MARVEL for improving query performance. The evaluation over adapted LUBM and SSB benchmark datasets show that evaluating queries over materialized views is on average 3-11 times faster than evaluating the queries over raw data.

In this thesis, we concentrated on optimizing the performance of analytical SPARQL queries. We intended to optimize the performance of aggregate queries in a federated setup since more and more data on the Web are interlinked. Thus, Chapters 3 and 4 investigated analytical query performance optimization in federated setup. On the other hand, optimizing the performance of aggregate SPARQL queries in a federation can not be achieved without optimizing the performance of aggregate queries on a single endpoint. Therefore, in Chapter 5, we concentrated on optimizing the performance for aggregate queries on standalone endpoints. In Chapter 6, we conclude our investigation and define the outlook for future research.

3 Structure of the Thesis

The thesis is organized as a collection of individual papers. Each chapter is self-contained and can be read separately. There may be some overlap of concepts, examples, and texts in the introduction and the section related to used notations in Chapters 4 and 5 as they are formulated in relatively similar kind of settings and remain in each chapter for self-containment. The chapters have been modified during the integration to include additional materials and experiments. Additionally, the bibliographies of each chapter have been combined into one, and references to “this paper” have been changed to references to “this chapter”.

The papers included in this thesis are listed in the following. Chapter 2 is based on Paper 1, Chapter 3 is based on Paper 2, Chapter 4 is based on Paper 3, and Chapter 5 is based on Paper 4.

1. Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. Towards Exploratory OLAP Over Linked Open Data - A Case

Study. In *International Workshops on Enabling Real-Time Business Intelligence (BIRTE), Riva del Garda, Italy, 2013, and Hangzhou, China, 2014, Revised Selected Papers*, pages 114–132.

2. Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. Processing Aggregate Queries in a Federation of SPARQL Endpoints. In *12th European Semantic Web Conference, (ESWC 2015), Portoroz, Slovenia*, pages 269–285, 2015

3. Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. Efficient Support of Analytical SPARQL Queries in Federated Systems. *In preparation for a conference submission*

4. Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. Optimizing Aggregate SPARQL Queries Using Materialized RDF Views. In *15th International Semantic Web Conference, (ISWC 2016) Kobe, Japan*, pages 341–359(1), 2016

Chapter 2

Towards Exploratory OLAP over Linked Open Data – A Case Study

Dilshod Ibragimov, Torben Bach Pedersen, Katja Hose and Esteban Zimányi

The paper has been published in the
Proceedings of the International Workshops on Enabling Real-Time Business Intelligence, BIRTE 2013, Riva del Garda, Italy, and BIRTE 2014, Hangzhou, China, pp. 114–132, 2015. DOI: 10.1007/978-3-662-46839-5_8
The final publication is available at Springer via `http://dx.doi.org/10.1007/978-3-662-46839-5_8`.

Ibragimov D., Hose K., Pedersen T.B., Zimányi E. Towards Exploratory OLAP Over Linked Open Data – A Case Study. In: Castellanos M., Dayal U., Pedersen T., Tatbul N. (eds) Enabling Real-Time Business Intelligence. Lecture Notes in Business Information Processing, vol 206, 2015.
The layout of the paper has been revised.

Abstract

Business Intelligence (BI) tools provide fundamental support for analyzing large volumes of information. Data Warehouses (DW) and Online Analytical Processing (OLAP) tools are used to store and analyze data. Nowadays more and more information is available on the Web in the form of Resource Description Framework (RDF), and BI tools have a huge potential of achieving better results by integrating real-time data from web sources into the analysis process. In this chapter, we describe a framework for so-called exploratory OLAP over RDF sources. We propose a system that uses a multidimensional schema of the OLAP cube expressed in RDF vocabularies. Based on this information the system is able to query data sources, extract and aggregate data, and build a cube. We also propose a computer-aided process for discovering previously unknown data sources and building a multidimensional schema of the cube. We present a use case to demonstrate the applicability of the approach.

1 Introduction

In the business domain, there is a constant need to analyze big volumes of information for intelligent decision making. Business intelligence tools provide fundamental support in this direction. In general, companies use data warehouses to store big volumes of information and OLAP tools to analyze it. Data in such systems are generated by feeding operational data of enterprises into data warehouses. Then, OLAP queries are run over data to generate business reports. Multidimensional Expressions (MDX) query language is the de-facto standard for OLAP querying.

Traditionally, such analyses are performed in a "closed-world" scenario, based only on internal data. With the advent of the Web, more and more data became available online. These data may be related to, for example, the market, competitors, customer opinions (e.g., tweets, forum posts), etc. Initially, these data were not suitable for machine processing. Later, a framework that extends the principles of the Web from documents to data converting the Web of Documents into the Web of Data was proposed. According to the standards, to facilitate a discovery of the published data, these data should comply with the Linked Data principles [1]. RDF was chosen as a standard model for data interchange on the Web [21]. With these principles in action, the whole Internet may be considered as one huge distributed dataspace.

With data being publicly available, businesses see the benefits of incorporating additional, real-time data into the context of information received from data warehouses or analyzing these data independently. Companies may explore new data opportunities and include new data sources into business analyses. A new type of OLAP that performs discovery, acquisition, integration, and analytical querying of new external data is necessary. This

type of OLAP was termed *Exploratory* OLAP [5].

In the past years the scientific community has been working on bringing these new BI concepts to end-users. The main focus of research was providing an easy and flexible access to different data sources (internal and external) for non-skilled users so that the users can express their analytical needs and the system is able to produce data cubes on-demand. Optimally, the internal complexity of such systems should be transparent to end-users.

In [22] a vision to new generation BI and a framework to support self-service BI was proposed. The process, according to this framework, is divided into several steps and consists of query formulation, source discovery and selection, data acquisition, data integration, and cube presentation phases. Based on this framework, we propose our approach to performing exploratory OLAP over Linked Open Data (LOD). For the sake of simplicity, our scenario considers only data available in RDF format and accessible over SPARQL endpoints [23].

The novel contribution of this chapter are:

– We define a multidimensional schema of an OLAP cube exclusively in RDF. This multidimensional schema allows to define remote data sources for querying during the OLAP analysis phase.

– We propose a computer-aided approach to deriving the schema of the OLAP cube from previously unknown sources.

The remainder of the chapter is structured as follows: in Section 2, we introduce a case study for exploratory OLAP scenario where the multidimensional schema and sources of data are already known. We show how we can retrieve data and build an OLAP cube. In Section 3, we propose ideas for sources discovery and schema generation for such cases. In Section 4, we present a conceptual framework for achieving exploratory OLAP over LOD. In Section 5, we discuss the related work. Finally, in Section 6, we conclude this chapter and identify future work.

2 A Movie Case Study

This scenario is based on the dataset originating from the Linked Movie Database[1] (LinkedMDB) website, which provides information about movies. LinkedMDB publishes Linked Open Data for movies, including a large number of interlinks to several datasets on the LOD cloud and references to related webpages. Data can be queried using a SPARQL endpoint[2].

[1] `http://data.linkedmdb.org`

[2] `http://data.linkedmdb.org/sparql`

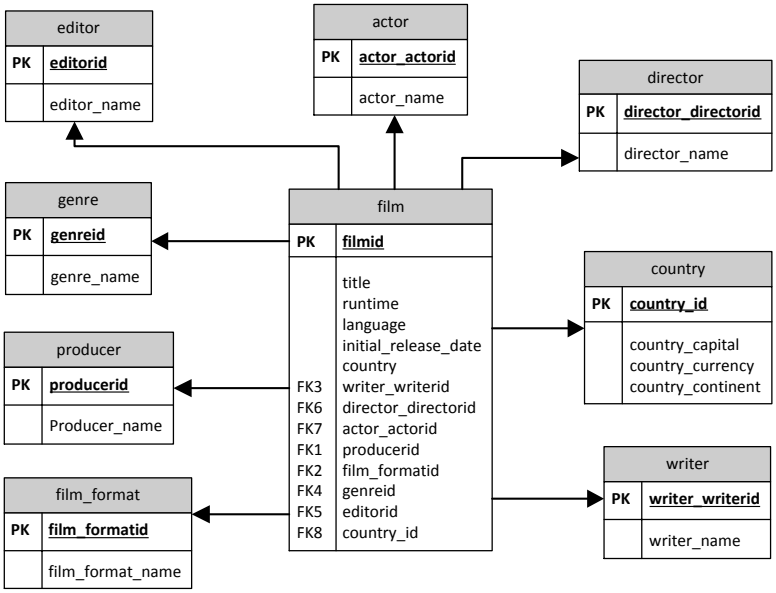


Fig. 2.1: Partial LinkedMDB logical schema

A typical movie record contains information about the movie, the actors who played in the movie, the director of the movie, the genre, the initial release date, the runtime, the country where it was produced, etc. An example record for the movie "The Order"[3] stored in LinkedMDB is as follows (all the prefixes used in the chapter are listed in the appendix):

```
<http://data.linkedmdb.org/resource/film/1005> rdf:type movie:film ;
  movie:actor <http://data.linkedmdb.org/resource/actor/32063> ;
  movie:actor <http://data.linkedmdb.org/resource/actor/42288> ;
  foaf:based_near <http://sws.geonames.org/2921044/> ;
  movie:country <http://data.linkedmdb.org/resource/country/DE> ;
  dc:date ‘‘2003,2003-09-05’’ ;
  movie:director <http://data.linkedmdb.org/resource/director/9091> ;
  movie:film_cut <http://data.linkedmdb.org/resource/film_cut/15031> ;
  movie:filmid ‘‘1005’’^^xsd:int ;
  movie:genre <http://data.linkedmdb.org/resource/film_genre/28> ;
  movie:initial_release_date ‘‘2003,2003-09-05’’ ;
  rdfs:label ‘‘The Order’’ ;
  movie:language <http://www.lingvoj.org/lingvo/en> ;
  foaf:page <http://www.imdb.com/title/tt0304711> ;
  movie:runtime ‘‘102’’ ;
  dc:title ‘‘The Order’’ .
```

A partial logical schema of the LinkedMDB is given in Figure 2.1. LinkedMDB also contains links to other datasets using the property `owl:sameAs`. For

[3] `http://data.linkedmdb.org/resource/film/1005`

example, a country information is interlinked to GeoNames[4]. Based on the analysis of GeoNames, the partial logical schema of GeoNames is illustrated in Figure 2.2.

Suppose a user wants to analyze data about movies. Examples of typical queries could be:

- Average runtime for movies by movie director and country
- Number of movies by continent and year

N.B.: She may want to do it in the context of information retrievable from GeoNames.

For this purpose, the user may want to construct a virtual data cube. Data will be retrieved from two sources: LinkedMDB and GeoNames. The data cube is considered virtual because data are not materialized in the local system. This data cube accepts user queries, queries the data sources, retrieves the information, processes it, and answers user queries. The multidimensional schema of such a data cube is given in Figure 2.3. The schema describes the dimensions: `Country (Population, Country Name)`, `Release Date (Year, Quarter, Month)`, `Director`, `Actor`, `Script Writer` and the measure: `Runtime`.

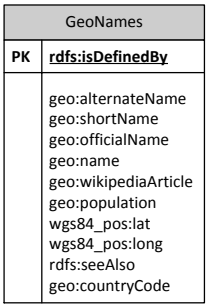


Fig. 2.2: Partial GeoNames logical schema

Knowing the structure of the cube, a user wants to find the average runtime for movies by director and country. She issues an MDX query as shown in Listing 2.1:

```
WITH MEMBER Measures.AvgRuntime AS Avg(Film.Director.CurrentMember, Measures.Runtime)
SELECT NON EMPTY {Film.Director.Members} ON COLUMNS,
       NON EMPTY {Film.Country.Members} ON ROWS
FROM [MoviesDataWarehouse] WHERE (Measures.AvgRuntime);
```

Listing 2.1: MDX Query on the Data Cube

Data on the Web are mostly stored and retrieved as RDF and not as relational data. Therefore, we propose to use a fully RDF-based approach for exploratory OLAP over LOD sources and to analyze data without converting them to relational data and storing them in a local data warehouse. Additionally, loading and storing highly volatile, real-time data in a local system may not be practical.

[4] `http://www.geonames.org/`

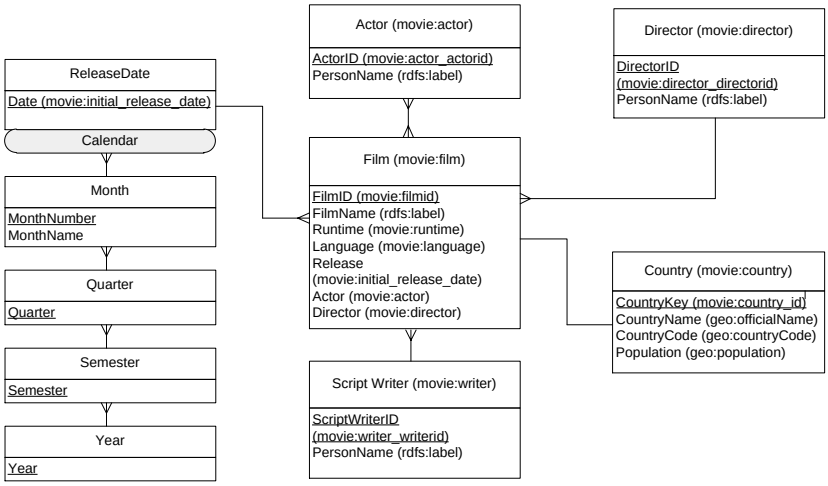


Fig. 2.3: Conceptual schema of the data cube

In our case study we use RDF vocabularies such as QB4OLAP [9] and VoID [10] to describe the multidimensional schema. QB4OLAP is an RDF vocabulary that allows the publication of multidimensional data. QB4OLAP can represent dimension levels, level members, rollup relations between levels and level members, etc. QB4OLAP can also associate aggregate functions to measures. VoID is an RDF Schema vocabulary for expressing metadata about RDF datasets. The vocabulary may specify how RDF data can be accessed using various protocols. For example, the SPARQL endpoint location can be specified by the property `void:sparqlEndpoint`. Based on the information from the multidimensional schema, the system will be able to identify the sources and query them. An excerpt of the multidimensional schema for our running example, expressed in the QB4OLAP and VoID vocabularies, is given in Listing 2.2.

```
## Data structure definition and dimensions
exqb:FilmCube a qb:DataStructureDefinition ;
    void:sparqlEndpoint
        <http://data.linkedmdb.org/sparql> ;
## Dimensions
qb:component [qb:dimension exqb:Actor];
qb:component [qb:dimension exqb:ReleaseDate];
qb:component [qb:dimension exqb:Director];
qb:component [qb:dimension exqb:Country];
## Definition of measures
qb:component [qb:measure exqb:Runtime];
## Attributes
qb:component [qb:attribute exqb:FilmName] .
```

```
## Dimension Properties and Hierarchies
exqb:year a qb4o:LevelProperty ;
    skos:closeMatch db:Year ;
    rdfs:comment "Film release year"@en ;
    qb4o:inDimension exqb:ReleaseDate .
exqb:quarter a qb4o:LevelProperty ;
    rdfs:comment "Film release quarter"@en ;
    qb4o:inDimension exqb:ReleaseDate .
exqb:ReleaseDate a qb:DimensionProperty .
exqb:Actor a qb:DimensionProperty ;
    skos:mappingRelation movie:actor ;
    rdfs:seeAlso owl:sameAs ;
    qb4o:hasAttribute exqb:PersonName .
```

Listing 2.2: Multidimensional Schema Expressed in QB4OLAP

To answer the MDX query, the system needs to send SPARQL queries to remote data endpoints for data retrieval. To do this, it first finds appropriate information for the measures and the dimensions specified in the MDX query from the multidimensional schema. The system finds the sources of data for dimensions /measures (`void:sparqlEndpoint`), all the attributes (`qb4o:hasAttribute`), the mapping information to map these attributes to the source equivalents (`skos:mappingRelation`), etc. For instance, for the MDX query given in Listing 2.1 the system needs to find the information about the `Runtime` measure and the `Director` and the `Country` dimensions. Then, the system sends SPARQL queries to the LinkedMBD and the GeoNames SPARQL endpoints. The query that is sent to LinkedMDB to retrieve the information regarding dimensions, attributes, and measures is given in Listing 2.3.

```
### Retrieving attributes, dimensions, and measures
CONSTRUCT {
    ?movieUrl exqb:Runtime ?runtime . ?movieUrl exqb:FilmName ?movieName .
    ?movieUrl exqb:Country ?country . ?country owl:sameAs ?owlCountry .
    ?movieUrl exqb:Director ?directorID . ?directorID exqb:PersonName ?directorName .
} WHERE {
    ?movieUrl rdf:type movie:film . ?movieUrl movie:country ?country .
    ?country owl:sameAs ?owlCountry . ?movieUrl rdfs:label ?movieName .
    ?movieUrl movie:runtime ?runtime . ?movieUrl movie:director ?directorID .
    ?directorID rdfs:label ?directorName .
}
```

Listing 2.3: SPARQL Query to LinkedMDB

This query uses the `CONSTRUCT` clause to automatically create triples. These triples specify the dimension attributes and therefore can easily be copied to the final QB4OLAP structure. An excerpt from the result returned to the system for the query is as follows:

```
<rdf:Description rdf:about="http://data.linkedmdb.org/resource/film/930">
  <exqb:FilmName>Godfather</exqb:FilmName>
  <exqb:Director rdf:resource="http://data.linkedmdb.org/resource/director/448"/>
  <exqb:Country rdf:resource="http://data.linkedmdb.org/resource/country/IN"/>
  <exqb:Runtime>158</exqb:Runtime>
</rdf:Description>
<rdf:Description rdf:about="http://data.linkedmdb.org/resource/film/2939">
  <exqb:Director rdf:resource="http://data.linkedmdb.org/resource/director/10494"/>
  <exqb:Runtime>120</exqb:Runtime>
  <exqb:FilmName>Raincoat</exqb:FilmName>
  <exqb:Country rdf:resource="http://data.linkedmdb.org/resource/country/IN"/>
</rdf:Description>
<rdf:Description rdf:about="http://data.linkedmdb.org/resource/director/448">
  <exqb:PersonName>K. S. Ravikumar (Director)</exqb:PersonName>
</rdf:Description>
<rdf:Description rdf:about="http://data.linkedmdb.org/resource/director/10494">
  <exqb:PersonName>Rituparno Ghosh (Director)</exqb:PersonName>
</rdf:Description>
<rdf:Description rdf:about="http://data.linkedmdb.org/resource/country/IN">
  <owl:sameAs rdf:resource="http://sws.geonames.org/1269750/"/>
</rdf:Description>
```

Then, the data from GeoNames may be downloaded. In our running example the system uses the URI received from the "owlCountry" property and use it in the VALUES statement of the SPARQL query. We use a VALUES statement to group several arguments together in one query. Our goal is to send as few queries as possible. Since GeoNames does not have an associated SPARQL endpoint, the query is sent to the mirrored endpoint (http://lod2.openlinksw.com/sparql):

```
CONSTRUCT {
    ?s geo:countryCode ?o1 . ?s geo:name ?o2 . ?s geo:population ?o3 .
} WHERE {
    ?s geo:countryCode ?o1 . ?s geo:name ?o2 . ?s geo:population ?o3 .
    VALUES (?s){ (<http://sws.geonames.org/1149361/>) ... (<http://sws.geonames.org/1269750/>) }
}
```

Listing 2.4: SPARQL Query to GeoNames

This query returns information about the country's population, name, and code. A sample answer may look as follows:

```
<http://sws.geonames.org/1149361/>  geo:countryCode  "AF" ;
        geo:name  "Islamic Republic of Afghanistan" ;
        geo:population  "29121286" .
```

The data obtained from the two sources are merged into a QB4OLAP structure: all received facts may be stored as qb:Observation instances (in OLAP terminology this corresponds to facts indexed by dimensions), all dimension instances are stored as triples. The aggregated values for measures are computed based on the qb4o:AggregateFunction function type. A sample QB4OLAP structure is given in Listing 2.5:

```
<http://data.linkedmdb.org/resource/film/810> a qb:Observation;
  qb:dataSet exqb:MoviesDataWarehouse ;
  exqb:Director < http://data.linkedmdb.org/resource/director/8629> ;
  exqb:Runtime 188;
  exqb:Country < http://data.linkedmdb.org/resource/country/IN> .
http://data.linkedmdb.org/resource/film/930> a qb:Observation;
  qb:dataSet exqb:MoviesDataWarehouse ;
  exqb:Director < http://data.linkedmdb.org/resource/director/448> ;
  exqb:Runtime 158;
  exqb:Country < http://data.linkedmdb.org/resource/country/IN> .
<http://data.linkedmdb.org/resource/country/IN>
        exqb:CountryName "India" ;
        exqb:CountryCode "IN" ;
        exqb:Population "1173108018" .
<http://data.linkedmdb.org/resource/director/448>
        exqb:PersonName "K. S. Ravikumar (Director)" .
```

Listing 2.5: Observations in QB4OLAP

Table 2.1: Aggregated Values

	Great Britain	India	United States	Venezuela	Pakistan	Russia	Netherlands
Sally Potter (Director)	86						96
Robert Aldrich (Director)			88				
Román Chalbaud (Director)		93					
Roland Joffé (Director)				87			
Gerald Thomas (Director)	78		83				

In case the number of returned triples is large and cannot be handled by a SPARQL endpoint or transferred over the Internet, the system can send aggregate subqueries to the sources. The aggregation can be performed on the graph patterns used for joining several federated SPARQL subqueries. This will help to reduce the number of records for which the values from the endpoints will be transferred. For example, the following subqueries return aggregate values (left) and additional information (right) on the runtime of the movies by director and country. The results can be connected via the values of the `?owlCountry`.

```
SELECT AVG(?runtime) ?dirName ?owlCountry
WHERE {
  ?movUrl exqb:Runtime ?runtime .
  ?movUrl exqb:Country ?country .
  ?cntr owl:sameAs ?owlCountry .
  ?movUrl exqb:Director ?dirID .
  ?dirID exqb:PersonName ?dirName .
} GROUP BY ?dirName ?owlCountry
```

```
SELECT ?owlCountry ?code ?c_name ?pop
WHERE {
  ?owlCountry geo:countryCode ?code .
  ?owlCountry geo:name ?c_name .
  ?owlCountry geo:population ?pop
}
```

Listing 2.6: Aggregate and Informational Subqueries

The intermediate results of the execution of the subqueries may be stored in an in-memory table. Then, the results of the execution of the subqueries will be merged into the QB4OLAP structure. Based on these data, the computed aggregated values are returned back to a user of the system. A sample answer to the previous MDX query may look as shown in Table 2.1.

3 Source Discovery and Schema Building for Exploratory OLAP

In the case study introduced in Section 2 we assume that the data sources and the multidimensional schema of the OLAP cube are known. However, in reality the discovery of essential data sources is not a trivial task. Despite the fact that the publication of Linked Data has gained momentum in recent years, there is still no single approach on how these data should be published to be easily discoverable. We identified three potentially interesting data source discovery approaches for further investigation. In all three approaches

Table 2.2: Freebase Query Partial Results

?s	?l	?count
http://rdf.freebase.com/ns/m.02nsjl9	Film character	2001832
http://rdf.freebase.com/ns/film.film/character	Film character	1384754
http://rdf.freebase.com/ns/film.actor	Film actor	874840
http://rdf.basekb.com/ns/m.0jsg30	Film performance	673398
http://rdf.freebase.com/ns/film.film	Film	557505
http://rdf.freebase.com/ns/m.0jsg4j	Film actor	492249
http://rdf.freebase.com/ns/film.film_crew_gig	Film crew gig	456500
http://rdf.basekb.com/ns/m.02nsjl9	Film character	410669
http://rdf.basekb.com/ns/m.0jsg4j	Film actor	215777
http://rdf.freebase.com/ns/m.02_6zn1	Film crewmember	205938

described below we show how we can derive a schema of the OLAP cube for the scenario discussed in Section 2.

3.1 Querying Knowledge Bases

The first approach is querying large knowledge bases such as DBpedia[5], Yago[6], or Freebase[7] to find relevant information. Data from such knowledge bases are usually freely accessible over SPARQL endpoints. Querying these endpoints for the term of interest may lead to the discovery of useful sources of data or the necessary information itself. Since the number of answers that come from these sources may be extremely large and not always relevant, there is a need for filtering the answers. Also, since the user entry may be ambiguous due to the ambiguity and complexity of natural languages, the end user needs to guide the process of source discovery by selecting most appropriate alternatives for further investigation.

To find some relevant information about the term "Film", we can send the following SPARQL query to the Freebase SPARQL endpoint:

```
SELECT ?s ?l COUNT(?s) as ?count
WHERE {
    ?someobj ?p ?s .  ?s rdfs:label ?l .
    FILTER(CONTAINS(?l, "Film") && (lang(?l) = 'en') && (!isLiteral(?someobj))) .
} ORDER BY DESC(?count) LIMIT 20
```

This query is optimized to allow sorting by relevance using the COUNT function so that the user sees the most relevant answers first. The partial result of the query is given in Table 2.2.

By examining the returned answer, the user may find some interesting triples and may want to explore these triples further. The system at this stage

[5] http://dbpedia.org/About

[6] http://www.mpi-inf.mpg.de/yago-naga/yago/

[7] http://www.freebase.com/

Table 2.3: Freebase Movie Instances

?s	?p	?o
http://rdf.freebase.com/ns/m.0pj5t	rdfs:label	Falling Down
http://rdf.freebase.com/ns/m.0swhj	rdfs:label	A Charlie Brown Christmas
http://rdf.freebase.com/ns/m.0m2kd	rdfs:label	Stand by Me
http://rdf.freebase.com/ns/m.07cz2	rdfs:label	The Matrix
http://rdf.freebase.com/ns/m.0c296	rdfs:label	Amélie
http://rdf.freebase.com/ns/m.0prk8	rdfs:label	Hamlet
http://rdf.freebase.com/ns/m.0j90s	rdfs:label	Guess Who's Coming to Dinner
http://rdf.freebase.com/ns/m.02yxx	rdfs:label	Fearless
http://rdf.freebase.com/ns/m.0p9rz	rdfs:label	Romeo and Juliet
http://rdf.freebase.com/ns/m.0symg	rdfs:label	Dead Man

helps the user to do so. For example, several triples should be retrieved for further exploration. In our case, one of the triples has a subject equal to <http://rdf.freebase.com/ns/film.film>. The following query returns several instances related to the triple pattern of interest:

```
SELECT ?s ?p ?o
WHERE {
    ?s ?p ?o . ?s ns:type.object.type ns:film.film . FILTER (lang(?o) = 'en').
} LIMIT 10
```

The result of the execution of the query is given in Table 2.3.

If the user decides that the selected samples satisfy the needs, the user is aided in building a multidimensional model of the OLAP cube. Our proposition for building a graph representation of the source is based on characteristic sets (CS) (Neumann and Moerkotte [14]), which contain the properties of RDF data triples for triple subjects. The system should also offer possible candidates for measures, dimensions, and dimensional attributes, identifying all triples related to the instances, their data types, etc. Then the user chooses the schema that most closely reflects the needs or directs the system for further search. In our example, the user may encounter properties of interest such as runtime, director, actors, and country by exploring the instance properties of the class ns:film.film:

```
ns:m.0c296        ns:film.film.country    ns:m.0345h;
ns:film.film.directed_by    ns:m.0k181;
ns:film.film.edited_by    ns:m.07nw1y6;
ns:film.film.genre    ns:m.05p553;
ns:film.film.initial_release_date    "2001-04-25"^^xsd:datetime;
ns:film.film.runtime..film.film_cut.runtime    ns:122.0;
ns:film.film.starring..film.performance.actor    ns:m.01y9t4;
ns:film.film.starring..film.performance.actor    ns:m.0jtcpc;
```

The whole process of discovering the sources and building the multidimensional schema needs to be guided by a user.

3.2 Querying Data Management Platforms

The second approach for source discovery is querying so-called data management platforms. One such platform is the Datahub[8]–the platform based on the CKAN[9] registry system. CKAN is an open source registry system that allows storing, distributing, and searching of the contents for spreadsheets and datasets. Search and faceting features allow users to browse and find the data they need. CKAN provides an API that can be used for searching the data by applications. For instance, CKAN's Action API provides functions for searching for packages or resources matching a user query. Using the Action API, we can list all the datasets (packages) residing in the system (`http://datahub.io/api/3/action/package_list`), view the dataset descriptions (`http://datahub.io/api/3/action/package_show?id=linkedmdb`), or search for datasets matching the search query (`http://datahub.io/api/3/action/package_search?q=Film`). The answer is returned in JSON format.

Querying the Datahub for a "Film" string returns 99 results, where 5 results have SPARQL endpoints: Prelinger Archives (`http://api.kasabi.com/dataset/prelinger-archives/apis/sparql`), Linked Movie Database (`http://data.linkedmdb.org/sparql`), DBpedia-Live (`http://live.dbpedia.org/sparql`), Europeana Linked Open Data (`http://europeana.ontotext.com/sparql`), and DBpedia (`http://dbpedia.org/sparql`). By retrieving several instances of triple patterns and identifying corresponding properties (the same process as proposed for querying knowledge bases), we may define the multidimensional schema needed for the OLAP cube.

3.3 Querying Semantic Web Search Engines

The third approach for sources discovery is querying semantic web search engines. An example of such search engines is Sindice[10], which also provides a Search API (`http://sindice.com/developers/searchapiv3`) using a query language (`http://sindice.com/developers/queryLanguage`). The Search API provides programmatic access to search capabilities of the search engine and returns the result in one of three formats: JSON, RDF, or ATOM. This API supports a keyword search to facilitate the discovery of relevant documents that contain either a keyword or a URI. The query language supports filtering the search results by URL, domain, class, predicate, ontology, etc. and grouping the search results by datasets.

Querying Sindice for the "Film" string returns many results (582,883),

[8] `http://datahub.io`

[9] `http://ckan.org/`

[10] `http://sindice.com/`

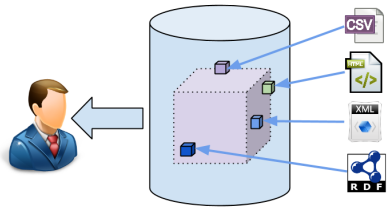


Fig. 2.4: Functional View

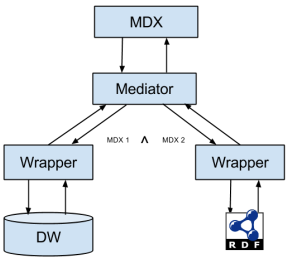


Fig. 2.5: Data Integration

mostly individual triples, but grouping the results by datasets allows identifying the datasets for further exploration. The following query to Sindice reveals the Linked Movie Database dataset (`http://data.linkedmdb.org`) for further exploration among others: `http://api.sindice.com/v3/search?q=Film&format=json&fq=format%3ARDF&page=6&facet.field=domain`

After discovering a proper source of information, we should apply the process of building the multidimensional schema of the OLAP cube.

4 Conceptual Framework

The main functionality of an exploratory OLAP system is illustrated in Figure 2.4. Here we assume that there may (optionally) exist some internal data depicted as a cube with dotted lines. These data may serve as a foundation for further exploration. A user may want to enrich/supplement these data by external data from the Web. Ideally, the system should be able to retrieve data stored in any format (HTML, XML, CSV, RDF, etc.). In Figure 2.4 these data are depicted as small colored cubes which extend the internal cube. This requirement imposes additional complexity over the system, so the part of the system that is responsible for exploratory OLAP can be further subdivided into several subparts, each handling another data format. In this chapter we concentrate on Linked Open Data and we describe our vision on how to achieve exploratory OLAP over Linked Open Data.

The envisioned architecture for the exploratory OLAP over Linked Open Data system is sketched in Figure 2.6. The system consists of four main modules. The Global Conceptual Schema module contains information about the schema of the specified data cube. In particular, it contains information about the measures, the dimensions and hierarchies in the dimensions, the potential aggregation functions over the measures, and pointers to data sources where the data are located. To represent this information, we propose to use the combination of QB4OLAP and VoID vocabularies. QB4OLAP allows

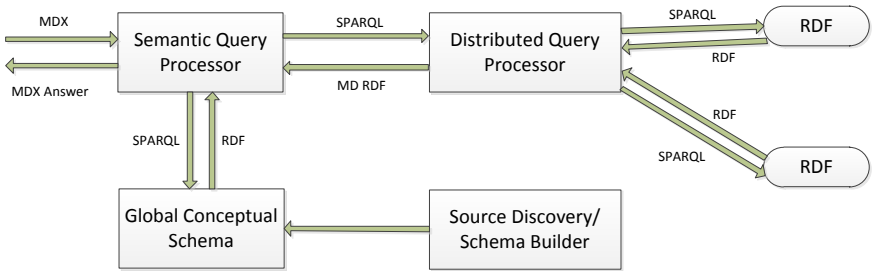


Fig. 2.6: System Architecture

defining dimensions, measures, and aggregations. The access and linkset metadata sections of the VoID vocabulary allow to describe data sources. An example of the multidimensional schema expressed in QB4OLAP that is part of the Global Conceptual Schema module can be found in Listing 2.2.

This combination of vocabularies is robust with respect to the schema complexity, the number of data sources, and the data volume. The schema complexity is handled by the QB4OLAP vocabulary as demonstrated in [24]. Recent changes to the QB4OLAP vocabulary [25] aid in defining complex multidimensional schemas with different hierarchies of levels in dimensions (balanced, recursive, ragged, many-to-many), different cardinalities between level members (one-to-many, many-to-many, etc), levels belonging to different hierarchies, etc. A number of data sources can be referenced in a multidimensional schema of a data cube with the help of the VoID vocabulary. Regarding the data volume, recent experiments show that triple stores per se are not worse for analytical queries than RDBMS [26], so we expect our approach to be sufficiently scalable.

The Semantic Query Processor is a module of the system that accepts an MDX query as input and produces a multidimensional SPARQL query using the QB4OLAP vocabulary for further processing. For this purpose, it queries the Global Conceptual Schema to find appropriate information – the measures and the dimensions specified in the MDX query. After having received the requested information, the Semantic Query Processor will formulate SPARQL queries to all data endpoints and send these queries to the Distributed Query Processing module for data retrieval. Examples of such SPARQL queries can be found in Listings 2.3 and 2.4. The Distributed Query Processor in turn queries all data endpoints, collects and merges all data, and returns the result back to the Semantic Query Processor (Listing 2.5). The returned answer is then either displayed to the user or passed to the calling module for the integration with data from the internal data warehouse.

The integration of external dimensional data with an internal data warehouse has been studied before. For instance, Pedersen et al. [27] present an

approach to the logical federation of OLAP and XML data sources. Following the same pattern, we envision that the system will have the mediator/wrappers to split and translate initial MDX query to other query languages. This is a common approach in distributed database systems [11]. The results received from the wrappers will then be merged by the mediator and shown to the user. The data integration architecture is depicted in Figure 2.5.

The Source Discovery/Schema Builder module is responsible for deriving a schema of the OLAP cube based on the user requirements. This module interacts with the user during the schema construction phase. The user specifies the domain/key concept of interest; the module searches for appropriate data sources and proposes the most relevant of them to the user. The module uses the approaches described in Section 3 to find interesting data sources. We propose to use all three approaches because none of these approaches alone guarantees full reliability. After identifying data sources, the system proposes a list of potential facts, dimensions, and measures, constructs possible multidimensional schemas, and presents them to the user for confirmation. This multidimensional schema is then used in the Global Conceptual Schema module.

5 Related Work

In the following, we review previous research in semantic web warehousing, source discovery, and distributed SPARQL query processing.

5.1 Semantic Web Data Warehousing

Related work for semantic web data warehousing can be divided into two categories. In the first category of approaches, the data is loaded into a local data warehouse that is built over a relational database management system. The schema of the data warehouse is generally determined by an administrator of the system and the data from the Linked Data sources are loaded into the defined tables. Then, the OLAP queries are run against the data stored in a star or snowflake schema. In the second category of approaches the OLAP operations are executed directly over RDF stores via SPARQL.

Determining schema information for a discovered data source helps in building a multidimensional model of a data cube. In an RDF dataset, the subjects that share the same properties can be grouped together. The result is a list of property sets with associated subjects. These property sets are called Characteristic Sets. Neumann et al. [14] used the knowledge about these sets for the estimation of the result cardinality for join operations in triple stores. In comparison, we instead employ characteristic sets as a basis for building a multidimensional data cube schema.

Romero et al. [16] defined a semi-automatic general-purpose method of building a multidimensional schema for a data warehouse from a domain ontology. The method aims to propose meaningful multidimensional schema candidates. The method defines main steps that lead to identifying facts, dimensions, and dimension hierarchies. The system is semi-automatic in the sense that it expects a user confirmation for suggested concepts proposed as potential facts. Once the user selects a concept as a fact concept, it will give rise to a multidimensional schema. The disadvantage of this approach is the requirement to have a corresponding domain ontology. This may not be the case for all data sources.

Similarly, a semi-automatic method for the identification and extraction of data expressed in OWL is defined in [28]. OWL/DL is used to transfer valid data into fact tables and to build dimensions. According to the proposed method, an analyst defines a multidimensional star schema based on the known ontology of the source of data. Then, the data from the sources are loaded into the data warehouse. Overall, this method does not allow populating a multidimensional schema with semantic web data from the newly discovered sources with previously unknown structures.

A framework to streamline the ETL process from Linked Open Data to a multidimensional data model is proposed in [13]. In contrast to [28], this work does not require previous knowledge and an ontology to collect the data. The data that are retrieved from Linked Open Data sources are first stored in an intermediate storage, where these data are partitioned based on the type. Then, the analyst investigates the tables and chooses measures and dimensions for the multidimensional data model. Afterwards, the system generates the schema for the fact table, selects dimensions, and dumps data into relational tables for performing OLAP analysis. The disadvantage of this method is the requirement to have a high-level analyst for intermediate result investigation and multidimensional schema construction.

The approach proposed in [29] uses an ETL pipeline to convert statistical Linked Open Data into a format suitable for loading into an open-source OLAP system. The data are presented using the RDF Data Cube (QB) vocabulary [30] suitable for statistical data. The data that are stored in a QB file are loaded, via an ETL process, into the data warehouse. Then, the OLAP queries can be executed over the data. The advantage of the data stored as QB is that the measures and dimensions are already partly defined, so the transformation of data into the multidimensional model is easier. However, the method is not suitable for data expressed in other RDF vocabularies.

The execution of OLAP queries directly over an RDF store is explained in [31]. Statistical data defined with the help of on RDF Data Cube (QB) vocabulary are used. These data are loaded to a triple store. OLAP queries are translated to SPARQL queries and are run over the triple store. However, the proposed approach is applicable only to the data presented in QB. More-

over, the observations in the data should not include any aggregated values, otherwise the computation is incorrect.

In the majority of the current approaches [13, 28, 29] Linked Open Data are loaded into the relational tables of a data warehouse for further analysis. Our approach does not require a relational database for the OLAP analysis of web data. Additionally, our approach handles all types of RDF data unlike the proposal of [31], where only data stored as RDF Data Cubes (QB) are processed. Furthermore, our approach retrieves data from multiple sources whereas other approaches work with a single source of information at a time.

5.2 RDF Source Discovery

Heim et al. [12] propose an approach that automatically reveals relationships between two known objects in a large knowledge base such as DBpedia and displays them as a graph. They use properties in semantically annotated data to automatically find relationships between any pair of user-defined objects and visualize them. Although this approach is not relevant to source discovery the idea of searching through knowledge bases may be applicable to it.

Exploring Linked Data principles for finding data sources is proposed in [32]. One of these principles includes the usage of HTTP-based URIs as identifiers, which may be understood as a data link that enables the retrieval of data by looking up the URI on the Web. Hence, by exploring data during the query execution process one can obtain potentially relevant data for the system. However, this technique is less suitable for bulk retrieval of RDF data, which is needed for OLAP processing.

The publication of Linked Data as services is investigated in [33, 34]. The use of Web Services and Service Oriented Architecture (SOA) is explored in this work. SOA facilitates easier data exchange between parties. A key component of SOA is the service repository, which serves the purpose of publishing and discovering services for future use. Research on service repositories for Web Services were extensive but the approach did not receive widespread adoption and was discontinued later. The main problem was the lack of support for expressive queries to identify and automate the discovery and consumption of services [33]. To address this problem, researchers propose to semantically annotate service descriptions to aid automatic discovery. Unfortunately, this technology did not receive widespread adoption either. If such a universal registry for services that publish Linked Data is created, a discovery and consumption of Linked Data from previously unknown sources will become easier.

An architecture of creating an up-to-date database of RDF documents by involving user participation in discovery of semantic web documents is described in [35]. This database can be used by search engines and semantic

web applications to provide search and user-friendly services over the discovered documents. However, the service does not support discovery of SPARQL endpoints – this part of the process is left for future work. Scalability issues are not considered and are left for future as well.

Search engines for the semantic web [15, 36] index the semantic web by crawling RDF documents and offer a search API over these documents. Different search engines use different index types: some index triples/quads, some index RDF documents. These search engines create an infrastructure to support application developers in discovering relevant data by performing lookup using, for example, full-text search over the literals. In this chapter we propose to use semantic web search engines to support the discovery of SPARQL endpoints.

In this chapter we further enhance existing approaches. We elaborate on ideas from [13, 14, 16] to build a multidimensional schema from previously unknown RDF data sources. Moreover, we extend principles from [12, 15] for SPARQL endpoint discovery by grouping related results by datasets. For increased reliability in source discovery, we propose to employ a combination of approaches. Additionally, we target our approach to non-professional data analysts.

5.3 Indexing and Distributed Query Processing

As Linked Data are scattered over the Web, efficient techniques for distributed query processing become an important part of the system. Regarding distributed query processing over multiple SPARQL endpoints, several approaches and frameworks were proposed in the past years. In contrast to the systems for source discovery mentioned above, most systems for distributed query processing over SPARQL endpoints rely on the presence of pre-computed indexes or statistics to identify the relevance of sources [37–41] and only a few frameworks can avoid the need of pre-computed information [18]. Whereas most systems specialize in one type of data access, exploratory data access or SPARQL endpoints, hybrid systems propose handling different types of native access [42], often in combination with local caching [43].

In addition to determining the relevance of sources for a given SPARQL query based on the binary decision whether a source provides data that is relevant to answer any part of a query, sources can be selected based on their benefit [44]. In doing so, additional aspects are considered such as the overlap of the data provided by available sources. As a result, the minimum number of sources that still produce the complete answer to the query can be selected.

6 Conclusions and Future Work

In this chapter, we presented a framework for exploratory OLAP over LOD sources. We introduced a system that uses a multidimensional schema of the data cube expressed in QB4OLAP and VoID. Based on this multidimensional schema, the system is able to query data sources, extract and aggregate data, and build an OLAP cube. We proposed to store multidimensional information retrieved from external sources in a QB4OLAP structure. We also introduced a computer-aided process for discovering previously unknown data sources necessary for the given data cube and building a multidimensional schema. We presented a use case to demonstrate the applicability of the proposed framework. In the future, we plan to finish the prototype of the proposed framework and test the solution on large-scale case studies.

A Prefixes Used in the Chapter

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX exqb: <http://example.org/exqb#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX movie: <http://data.linkedmdb.org/resource/movie/>
PREFIX lmdbres: <http://data.linkedmdb.org/resource/>
PREFIX geo: <http://www.geonames.org/ontology#>
PREFIX wgs84_pos: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX qb: <http://purl.org/linked-data/cube#> .
PREFIX qb4o: <http://purl.org/olap#> .
PREFIX xml: <http://www.w3.org/XML/1998/namespace> .
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> .
PREFIX skos: <http://www.w3.org/2004/02/skos/core#> .
PREFIX foaf: <http://xmlns.com/foaf/0.1/> .
PREFIX dc: <http://purl.org/dc/elements/1.1/> .
PREFIX db: <http://dbpedia.org/resource/> .
PREFIX ns: <http://rdf.freebase.com/ns/> .
```

Chapter 3

Processing Aggregate Queries in a Federation of SPARQL Endpoints

Dilshod Ibragimov, Torben Bach Pedersen, Katja Hose and Esteban Zimnyi

The paper has been published in the
Proceedings of the 12th European Semantic Web Conference, ESWC 2015, Portoroz, Slovenia, pp. 269–285, 2015. DOI: 10.1007/978-3-319-18818-8_17
The final publication is available at Springer via `http://dx.doi.org/10.1007/978-3-319-18818-8_17`

The layout of the paper has been revised.

Abstract

More and more RDF data is exposed on the Web via SPARQL endpoints. With the recent SPARQL 1.1 standard, these datasets can be queried in novel and more powerful ways, e.g., complex analysis tasks involving grouping and aggregation, and even data from multiple SPARQL endpoints, can now be formulated in a single query. This enables Business Intelligence applications that access data from federated web sources and can combine it with local data. However, as both aggregate and federated queries have become available only recently, state-of-the-art systems lack sophisticated optimization techniques that facilitate efficient execution of such queries over large datasets. To overcome these shortcomings, we propose a set of query processing strategies and the associated Cost-based Optimizer for Distributed Aggregate queries (CoDA) for executing aggregate SPARQL queries over federations of SPARQL endpoints. Our comprehensive experiments show that CoDA significantly improves performance over current state-of-the-art systems.

1 Introduction

In recent years, we have witnessed the growing popularity of the Semantic Web and the Open Data movement. Today, there are still many open issues but we are constantly getting closer to making Tim Berners-Lee's vision of the Web of Data [45,46] become a reality that a broad spectrum of people can benefit from. The status of a major component of this vision is described by the Linked Open Data cloud[1], which has been growing rapidly and has now reached a size of 1014 nodes providing access to billions of triples. Hence, a plethora of data is available in RDF format [2], published as Linked Open Data [1], accessible free of charge, and often queryable via SPARQL [3] endpoints. Although much more data is available in plain RDF, the design issues [8] according to which Linked Open Data is published and especially the contained links to other datasets give these datasets great potential.

Building upon these standards in combination with the SPARQL 1.1 standard [47], novel applications can be built that are interesting to a broad range of users, including companies. With the data being publicly available, companies can integrate their private data with RDF datasets from the Web and enable analyses that were not possible before. A company might, for instance, be interested in analyzing its revenue in different countries against macro-economic indicators of these countries. As companies usually do not maintain such information locally, the missing information can be obtained from the World Bank[2] when needed. This data can even be accessed as

[1] `http://lod-cloud.net/`

[2] `http://www.worldbank.org/`

Linked Open Data (World Bank Linked Data[3]) and queried via a SPARQL endpoint. In doing so, the company has efficient access to up-to-date information without increasing costs of local maintenance. Furthermore, as the company is accessing Linked Data, obtaining more information (geographical, census, etc.) for further analyses can efficiently be retrieved from linked sources, such as GeoNames [48] and DBpedia [49].

Such analyses that companies are interested in, however, are based on complex queries that involve grouping and aggregation. Whereas these concepts have been available in SQL for a long time, it has only recently become possible to formulate such queries on RDF data using the extensions offered by the SPARQL 1.1 query language [7]. In addition, SPARQL 1.1 also standardizes Federated Queries [4] and thus facilitates the formulation of queries that involve multiple sources. Without this extension such queries need to be split up into subqueries that can be executed on the remote sources. Computing the final result needs to be done by the local system, e.g., computing an additional join that combines the partial results. With Federated Queries, the complete query can be formulated as a single SPARQL statement that a query processor can optimize before execution. Hence, all these extensions in combination make it finally possible to formulate, optimize, and execute analytical queries over federations of SPARQL endpoints automatically so that the user does not need to implement pre-processing of partial results locally but can concentrate on formulating the query.

But being able to formulate such queries is not enough, we also need sophisticated query optimization techniques that allow for evaluating such queries efficiently. The literature proposes techniques for subproblems, such as semantics and completeness of federated queries [17, 18, 50–52], source quality and selection [44, 53], etc. As the SPARQL 1.1 standard is not yet completely supported by all SPARQL endpoints [19], there is only little research regarding the evaluation of queries involving aggregation and grouping. To the best of our knowledge, this is the first work to investigate aggregate queries in the context of federations of SPARQL endpoints and their optimization. In summary, the contributions of this chapter are:

- the Mediator Join, SemiJoin, and Partial Aggregation query processing strategies for this scenario
- a cost model and techniques for estimating constants and result sizes for triple patterns, joins, grouping and aggregation
- the combination of these with the processing strategies into the Cost-based Optimizer for Distributed Aggregate queries (CoDA) approach for aggregate queries in federated setups that is generally able to choose the best execution strategy among a number of alternatives

[3] `http://worldbank.270a.info/`

- a comprehensive experimental evaluation showing that CoDA is efficient, scalable, and robust over different scenarios, and significantly faster than state-of-the-art triple stores

The remainder of the chapter is structured as follows. Section 2 discusses a motivating example and a preliminary analysis of state-of-the-art triple stores for aggregate queries in federations of SPARQL endpoints. Related work is discussed in Section 3. Section 4 identifies several alternative strategies for processing aggregated SPARQL queries in a federated setup. Section 5 introduces a cost-based query optimizer for aggregate queries over federations of SPARQL endpoints. The results of our evaluation are presented in Section 6; Section 7 concludes the chapter.

2 Motivating Example and Preliminary Analysis

In March 2011, an earthquake in the Pacific triggered a powerful tsunami and led to a huge devastation at the Japanese coast, which eventually caused a nuclear accident[4]. After these events had happened, the Ministry of Education, Sports, Culture, Science and Technology of Japan made daily announcements of radioactivity statistics observed hourly at 47 prefectures. These observations from March 16, 2011 to March 15, 2012 were converted to RDF data by Masahide Kanzaki and made publicly available via a SPARQL endpoint[5]. Listing 3.1 shows an example observation in RDF format.

```
#observation
<http://www.kanzaki.com/works/2011/stat/ra/
        20110414/p13/t08>
    rdf:value "0.079"^^ms:microsv ;
    ev:place <http://sws.geonames.org/1852083/> ;
    ev:time <http://www.kanzaki.com/works/2011/
        stat/dim/d/20110414T08PT1H> ;
    scv:dataset <http://www.kanzaki.com/works/
        2011/stat/ra/set/moe> .
#dimension - place
<http://sws.geonames.org/1852083/>
    vcard:region "Tokyo"@en ;
    vcard:locality "Shinjuku"@en ;
    gn:lat "35.69355" ;
    gn:long "139.70352" .
#dimension - time
<http://www.kanzaki.com/works/2011/stat/dim/d/
        20110414T08PT1H>
    rdfs:label "2011-04-14T08";
    tl:at "2011-04-14T08:00:00+09:00"
        ^^xsd:dateTime ;
```

[4] http://en.wikipedia.org/wiki/2011_T%C5%8Dhoku_earthquake_and_tsunami

[5] http://www.kanzaki.com/works/2011/stat/ra/ As this endpoint is not maintained by ourselves, we created our own SPARQL endpoint providing access to the same dataset: http://164.15.78.105:8890/sparql

```
    tl:duration "PT1H"^^xsd:duration .
```

Listing 3.1: Radioactivity Observation Example

The places that an observation was recorded at is represented by a URI from GeoNames [48]. Among other information, Geonames provides hierarchical information, i.e., every entity has a parent entity. Towns and villages in Japan, for example, have a district as parent. A district in turn has a prefecture as parent, and a prefecture has a country as parent. The following triple, for instance, encodes that the town "Nakayama" has "Higashimurayama" as its parent: `<http://sws.geonames.org/7450017/> gn:parentFeature <http://sws.geonames.org/2112760/>`

With the observations of radioactivity in multiple geographical locations (cities in our case) and information about their upper administrative divisions (prefectures in Japan) from GeoNames, interesting analyses become possible. For instance, we can compute the average radioactivity separately for each prefecture in Japan and use this information to find out which prefectures were more affected than others. Or we can compute the minimum and maximum radioactivity for each prefecture and hence identify the changes in radioactivity over the one-year observations. Formulating such queries involves grouping and aggregation (AVG, MIN, MAX, etc.) as well as combining information from two SPARQL endpoints. Listing 3.2 shows an example query that computes the average radioactivity for all prefectures in Japan. This query could be executed at a triple store with information about radioactivity and uses the LOD Cloud Cache SPARQL endpoint (`http://lod2.openlinksw.com/sparql`) to query GeoNames data remotely.

```
SELECT ?regName (AVG(?floatRV) AS ?average)
WHERE {
    ?s ev:place ?placeID . ?s ev:time ?time .
    ?s rdf:value ?radioValue .
    SERVICE http://lod2.openlinksw.com/sparql
    {
        ?placeID gn:parentFeature ?regionID .
        ?regionID gn:name ?regName .
    }
    BIND (xsd:float(?radioValue) as ?floatRV) .
}
GROUP BY ?regName
```

Listing 3.2: Aggregate Query over Radioactivity Observations

At first glance, this query does not seem very complicated. However, current state-of-the-art triple stores, such as Virtuoso v07.10.3207, Sesame v2.7.11, and Jena Fuseki v1.0.0 (based on ARQ) timed out while trying to answer this query. By analyzing the network traffic with the Wireshark[6] network protocol analyzer, we found out that Virtuoso and Fuseki are trying to send a query to the endpoint specified in the SERVICE clause for every single

[6] `http://www.wireshark.org`

radioactivity observation, while Sesame is trying to download all triples that match the patterns defined in the SERVICE subquery from the remote endpoint. In the first case, a triple store needs to send more than 400,000 requests to answer the query, and in the second case it needs to download more than 7.8 million triples from the remote endpoint. These strategies are obviously inefficient and need to be optimized.

3 Related Work

Federated query processing in database management systems (DBMS) has been a topic of research for several decades. In contrast to well-structured classic data models, federated RDF systems support arbitrary RDF datasets (even without explicit schema) and allow the use of special constructs to perform joins and express bindings (such as VALUES) not present in SQL-based systems.

The literature proposes a number of approaches for querying federated RDF sources. Some of these approaches require the availability of VoID [54] statistics. SPLENDID [40], for instance, uses VoID statistics to select a query execution plan for a federated query. For triple patterns not covered in the VoID statistics, the system requests the information by issuing SPARQL ASK queries. The system makes use of a cost-based model and cardinality estimations for selecting a query plan. However, the SPLENDID system and its cost-model do not cover the combination of grouping, aggregation, and SERVICE subqueries.

FedX [18] uses SPARQL ASK queries for triple patterns in a query to collect basic information that can be used for source selection. It implements bound joins with SPARQL UNION keyword (similar to a semi-join) to group triple patterns related to one source and, thus, reduces the number of queries that are sent. FedX has originally been developed based on the SPARQL 1.0 standard and does not use cost-based query optimization. Hence, it does not provide any particular optimization techniques for our use case and would always use a semi-join based strategy, which is only one of the options our optimizer (CoDA) chooses from.

ANAPSID [55] uses a catalog of endpoint descriptions to decompose a user query into subqueries that can be executed by separate endpoints. The query engine implements a technique based on the symmetric hash join [56] and the XJoin [57] to execute subqueries in a non-blocking fashion. SI-HJoin [58] also uses a hash join implementation to enable pipelining in combination with a lightweight cost-model with weight factors calibrated for remote systems. Both approaches were not designed with regard to aggregate queries and use a hash join implementation so that results from a join can already be forwarded to other operators in the query execution tree. How-

ever, pipelining is not helpful for analytical queries since the complete result of the query is needed for the aggregation.

Avalanche [59] and WoDQA [60], on the other hand, do not maintain data source registrations. Avalanche depends on third parties such as search engines to find a proper data source for executing a query. Statistics about cardinalities and data distributions are considered for breaking a query into a set of subqueries that in combination provide a full query answer. Then, these subqueries are executed in parallel against several endpoints. WoDQA uses VoID directories such as CKAN (`http://ckan.net`) and VoIDStore (`http://void.rbkexplorer.com`) to find possible sources of data. The system uses VoID statistics to group triple patterns into subqueries in a federated form and executes it by Jena ARQ.

An RDF data processing system that supports simple transactional queries as well as complex analytical queries is proposed in [61]. Aggregate queries are efficiently resolved by the system by using special look-up mechanisms. However, the system does not consider aggregate queries in a federated environment.

SPARQL-DQP [62] on the other hand, discusses semantics of the SPARQL 1.1 federation extension on a theoretical level and introduces the notion of well-defined patterns. It focuses on the optimization of federated queries in the presence of OPTIONAL subqueries but it was not designed to optimize and support analytical queries. Different strategies to implement federated queries in SPARQL 1.1 are discussed in [52]. Several limitations that may cause incorrect results and the potential validity restrictions are identified and fixes are proposed.

In summary, only very few approaches consider analytical queries [61,62] but not in the context of a federated setup. Most state-of-the-art approaches for federated query processing are designed with a focus on SPARQL 1.0 [18, 40,55,59,60] and lack full support of the more recent SPARQL 1.1 standard or do not offer support or particular optimizations for analytical queries. In contrast, this chapter proposes a cost-based approach to optimize and execute aggregate SPARQL queries over federations of endpoints.

4 Federated Processing of Aggregate Queries

In this section, we will systematically outline several strategies that can be used to evaluate aggregated queries in federations of SPARQL endpoints. Section 5 will then introduce a cost-based approach to choose the best strategy for a query.

For ease of presentation, this section focuses on queries with a single `SERVICE` subquery. But the discussed principles can be extended to the general case of well-designed patterns with strongly bound variables [50]. The

proposed approach can be combined with rule-based rewriting so that subpatterns, and especially joins, are evaluated in a cost-minimizing order. If an endpoint imposes limits on result sizes, then additional techniques, such as pagination [52], are used.

In the following, we use P_{AGG} to represent the original user query and P_e denotes the SERVICE subquery evaluated at SPARQL endpoint e. P_M represents the subquery that is created from the original query P_{AGG} by extracting P_e, adding a join on their common variables $var(P_e) \cap var(P_M)$, and, depending on the strategy, preserving grouping and aggregation. P_M is evaluated on the same endpoint M that P_{AGG} was sent to. Note that this section focuses on the implementation of the joins combining the partial results of the subqueries evaluated by remote endpoints. We do not make any restrictions on the local implementations that the remote endpoints use to evaluate joins contained in the subqueries they receive.

Mediator Join Strategy (MedJoin). The first strategy we describe is based on the mediator join technique that is used by many approaches for federated SPARQL query processing. The mediator/federator is the SPARQL engine that receives a query P_{AGG} from the user. The query optimizer at the mediator M defines P_e and P_M and sends P_e to endpoint e whereas P_M is processed on the endpoint m. Parallelization can be exploited by processing P_M and P_e at the same time. The main principle is to find all solutions to P_e and P_M first and then compute the remaining operations at the mediator, including the join (on `?placeID` in the example below) that combines the partial results as well as grouping and aggregation. Listings 3.3 and 3.5 illustrate P_M and P_e for our running example query (Listing 3.2).

```
SELECT ?placeID ?radioValue WHERE {
  ?s ev:place ?placeID; ev:time ?time.
  ?s rdf:value ?radioValue.
}
```

Listing 3.3: MedJoin: Query P_M

```
SELECT ?placeID ?regName WHERE {
  ?placeID gn:parentFeature ?regionID.
  ?regionID gn:name ?regName.
}
```

Listing 3.4: MedJoin: Query P_e

Note that due to the fact that SPARQL does not remove duplicate results, we do not need to keep all variables in the select clauses of P_e and P_M. If duplicates were removed (like in SQL), we would have to keep all variables in the subqueries to ensure that the number of tuples that form the result are preserved, otherwise the average function in our example query would not return the correct result.

In principle, constructs such as `OPTIONAL` and `FILTER` are assigned to the subqueries that their variables refer to. If there is a complex expression,

e.g., a FILTER is defined on a condition involving variables from different subqueries (e.g., $?a < ?b$), then the FILTER is evaluated after the partial results are combined at the mediator. The strength of this strategy is that partial queries can be evaluated in parallel. However, it can easily become expensive if the intermediate results are very large or when the datasets are very big.

Semi Join Strategy (SemiJoin). This strategy is based on the bound join or semi-join technique [17, 18], which was already available based on UNION or FILTER constructs in SPARQL 1.0. The recent SPARQL 1.1 standard, however, supports the VALUES clause, which allows for a much more elegant solution.

The main principle of this strategy is to execute the subquery with the smallest result first and use the retrieved results as bindings for the join variables in the other subquery. The intuition is that for selective joins, sending a few partial results to an endpoint is much faster than receiving the complete result for the more general subquery. It is then the task of the cost optimizer to identify the most promising order of execution of subqueries. Constructs, such as FILTER and OPTIONAL, can be assigned to subqueries as discussed for MedJoin. Let us consider an example query with a FILTER.

```
SELECT ?regName (AVG(?radioValue) AS ?average) WHERE {
  ?s ev:place ?placeID . ?s ev:time ?time . ?s rdf:value ?radioValue .
  SERVICE <http://lod2.openlinksw.com/sparql>{
    ?placeID gn:parentFeature ?regionID . ?regionID gn:name ?regName .
  } FILTER(?radioValue < 0.08) . } GROUP BY ?regName
```

Listing 3.5: MedJoin: Query P_e

This query can be evaluated efficiently by evaluating query P_M (Listing 3.6) and then using the obtained bindings for the join variable ?placeID in the VALUES clause of the query P_e (Listing 3.8).

```
SELECT ?placeID ?radioVal
WHERE {
  ?s rdf:value ?radioVal ;
  ev:place ?placeID; ev:time ?time.
  FILTER (?radioValue < 0.08) . }
```

Listing 3.6: SemiJoin: Query P_M

```
SELECT ?placeID ?regName
WHERE { ?placeID gn:parentFeature ?rgID.
  ?rgID gn:name ?regName.
  VALUES (?placeID) {
  <http://sws.geonames.org/1852083/>...} }
```

Listing 3.7: SemiJoin: Query P_e

In contrast to MedJoin, this strategy evaluates the subqueries sequentially and is particularly efficient for selective joins. However, as the VALUES clause is not yet widely supported by existing endpoints [19], the SPARQL 1.0 compliant alternatives of UNION (or FILTER) must often be used.

Partial Aggregation Strategy (PartialAgg). For queries where the grouping attributes of the original query contain a subset of the variables of the subquery that is executed first and the aggregate values are contained in the subquery that is evaluated second, further optimization is possible. The Partial Aggregation Strategy (PartialAgg) builds upon MedJoin by extending the subquery executed second with a `GROUP BY` clause and aggregate functions. The goal is to reduce the size of the partial result and compute partial aggregate values early so that P_{AGG} can be evaluated more efficiently.

Using PartialAgg our running example query (Listing 3.2) is decomposed into P_M (below) and P_e (Listing 3.8). First, P_M is computed, the result bindings are fed into the `VALUES` clause of P_e, and P_{AGG} combines the partial results via a join and computes final grouping and aggregation.

```
SELECT ?placeID (SUM(?radioValue) AS ?sum) (COUNT(?radioValue) AS ?count)
WHERE { ?s ev:place ?placeID; ev:time ?time; rdf:value ?radioValue . }
GROUP BY ?placeID
```

Listing 3.8: SemiJoin: Query P_e

Note that P_M here groups by `?placeID` whereas the original query (Listing 3.2) groups by `?regName`, this is because P_M uses the join attributes $var(P_e) \cap var(P_M)$ in the `GROUP BY` clause. Whereas a particular placeID would occur in many results for P_M in the MedJoin strategy, the additional grouping here guarantees that the result set contains only one. Hence, the size of the intermediate result is reduced.

When performing such an optimization, however, we need to take into account whether the aggregate function in the original query is algebraic or distributive [63]. Computing aggregates for distributive functions (SUM, MIN, MAX, COUNT) is straightforward, while for computing AVG we first need to compute both SUM and COUNT in separate and in the final step divide the sum of all intermediate SUMs by the sum of all intermediate COUNTs, i.e., $AVG = \frac{\sum_{i=1}^{N} SUM_i}{\sum_{i=1}^{N} COUNT_i}$.

5 Cost-Based Query Optimization

For each user query, the query optimizer needs to decide which of the strategies that we discussed in the previous section to use. In this section, we present CoDA (**Co**st-based Optimizer for **D**istributed **A**ggregate Queries). A cost-based optimizer finds the best strategy by computing query execution costs for different alternative query execution plans and choosing the one with minimum costs. In the remainder of this section, we first sketch how the query optimizer works, then we introduce the cost model. Finally, we present details of the cost model regarding cardinality estimation and processing costs.

5.1 Query Optimizer

To find the best query execution plan, we need to systematically examine alternative query execution plans that produce the same result. This process, called plan enumeration, and can easily become expensive if all possible alternative query plans are examined.

We first decompose the original query into multiple subqueries as described in Section 4. We obtain a local query P_L, endpoint queries $P_{e_1}, \ldots, P_{e_n}$, and the P_{AGG} that describes how to compute the aggregation and how to connect the other subqueries.

As a first step, we optimize the subqueries in separate, e.g., reordering the triple patterns based on a cost model for local execution so that the sizes of intermediate results and execution costs are minimized.

Afterwards, we enumerate all possible plans that combine these subqueries using the strategies introduced in Section 4. For each of these alternative plans, we estimate execution costs (as described in the remainder of this section) and choose the plan with the minimum costs to execute the query.

5.2 Cost Model

The overall costs of a distributed query execution plan C_T consist of the costs for communication between endpoints and mediator (communiation costs) C_C and the costs for processing the query on the data (processing costs) C_P:

$$C_T = C_P + C_C \tag{3.1}$$

Some parts of the query are executed locally, some remotely. Hence, we estimate C_T for each subquery in separate and compute the costs of the complete query plan by adding up the costs of its subqueries.

For subqueries that are executed in parallel, as for the MedJoin strategy, we need to consider parallel execution in our cost model. As the subquery that takes the longest time to execute determines the time when the result is available, we take the maximum instead of the sum in such cases, e.g., $C_T(S_1, S_2) = \max(C_T(S_1), C_T(S_2))$, where $C_T(S_i)$ denotes the costs of executing subquery S_i.

To estimate C_T, we need to define how to estimate C_C and C_L for a subquery. Depending on the subqueries in the query execution plan, this can easily become complex. The communication costs for retrieving the answer for subquery S_i from an endpoint can be estimated as:

$$C_C(S_i) = C_O + c_{S_i} \cdot C_{map} \tag{3.2}$$

where C_O denotes the overhead to establish communication, c_{S_i} denotes the estimated number of returned solution mappings for the subquery, and C_{map}

denotes the costs of transferring a single solution mapping.

We can also apply Equation 3.2 to compute the costs of sending a query containing solution mappings in the `VALUES` clause by using the number of passed solution mappings as c_{S_i}.

Processing costs are determined by I/O and CPU costs and are very specific to the particular triple store and available indexes, current load, hardware characteristics, implemented algorithms, etc. As such details are not available for endpoints, we estimate processing costs based on the amount of data that the query is evaluated on. We assume, however, that indexes are used to access triples matching a triple pattern efficiently. We obtain:

$$C_L = \sum_{t=1}^{M} (c_{tp} \cdot C_{IO}) \tag{3.3}$$

where c_{tp} is the estimated number of selected solution mappings for triple pattern t that is part of the subquery, and C_{IO} denotes the costs of processing a single triple. To evaluate aggregate subqueries, we add C_{AGG} to the processing costs:

$$C_{AGG} = c_{AGG} \cdot C_{AGG} \tag{3.4}$$

where c_{AGG} represents the number of result mappings that grouping and aggregation are evaluated on and C_{AGG} represents the costs for processing one of them.

5.3 Estimating Constants

The cost estimation formulas introduced above rely on several system-specific constants, i.e., C_O, C_{map}, andC_{AGG}. As the optimizer is running at the mediator, computing these values for the local system is much easier than for endpoints. Nevertheless, as each endpoint has different characteristics, we need to obtain estimates. CoDA estimates these values based on several probe queries. The estimates are reused for future queries and repeated regularly to account for changes at the endpoints.

C_{map} is estimated using queries such as: `SELECT * WHERE { ?s ?p ?o } LIMIT #L`. This query is executed several times with different values for `#L` and measured how long it takes to receive an answer from the endpoint. Based on the pairwise difference between the queries' execution times and `#L`, we estimate the average time for a single result C_{map}.

C_O is estimated based on queries that do not retrieve data from triple stores such as: `SELECT(1 AS ?v){}` or `ASK{}`. Multiple queries are executed to determine an average.

C_{AGG} is estimated based on queries such as: `SELECT COUNT(*) WHERE { ?s ?p ?o } GROUP BY #g`. Again, multiple queries with different valid values for `#g` and `#c` are used to build an average. By measuring the time it

takes to receive the results and substracting the message overhead C_O and the costs of transferring the result based con C_{map}, we can estimate C_{AGG}. Note that C_{AGG} represents the costs to process a single input triple. Hence, before computing the average over multiple queries, we need to divide by the number of triples that the aggregate query was computed on – this can conveniently derived from the query result (`COUNT(*)` is the number of input triples for each group).

Note that these estimates might not be perfectly accurate but this is acceptable for our purposes because we do not aim at accurately predicting the execution costs of a query execution plan but only to find out which execution plan is more efficient than the others.

5.4 Result Size Esimation

Another important part of the cost model is estimating the size of partial results (result cardinality). Similar to related work [40, 41], we base our estimations on VoID statistics [54, 64] as this is a standardized format and most commonly used. Nevertheless, not all SPARQL endpoints offer such statistics. Hence, it is somethings necessary to download a dump of the remote data to compute the statistics or send a series of SPARQL queries with `COUNT` functions to the endpoint.

VoID statistics can logically be divided into three parts: dataset statistics, property partition, and class partition. The dataset statistics describe the complete dataset: the total number of triples (`void:triples`, c_t), the total number of distinct subjects (`void:distinctSubjects`, c_s), and the total number of distinct objects (`void:distinctObjects`, c_o). The property partition contains such values for each property of the dataset ($c_{p,t}, c_{p,s}, c_{p,o}$). Finally, the class partition shows the number of entities of each class (`void:entities`).

Estimating Result Sizes for Basic Triple Patterns

To estimate result sizes for complex queries, we first need to estimate the result size of basic queries that consist of a single triple pattern and optionally a condition expressed by a `FILTER`.

Based on the void statistics, we can estimate the result size c_{res} of a triple patterns as follows: `(?s ?p ?o)` is directly given by c_t, `(s ?p ?o)` can be estimated as $\frac{c_t}{c_s}$, `(?s ?p o)` as $\frac{c_t}{c_o}$, and `(s ?p o)` as $\frac{c_t}{c_s \cdot c_o}$. If the predicate is specified in the triple pattern, we can estimate the result sizes as follows: `(?s p ?o)` is directly given by $c_{p,t}$, `(s p ?o)` can be estimated as $\frac{c_{p,t}}{c_{p,s}}$, `(?s p o)` as $\frac{c_{p,t}}{c_{p,o}}$. , and `(s p o)` as $\frac{c_{p,t}}{c_{p,s} \cdot c_{p,o}}$. Tighter estimates based on VoID statistics are possible when the property `rdf:type` is used [41].

We further introduce several optimizations that are often used in relational database systems [65]. As distributions are skewed, we assume a Zip-

fian distribution of values and multiply c_{res} with the correction coefficient of 1.1 (close to Zipfian ideal). In case a `FILTER` involves an inequality comparison (e.g. $?x >= 10$), we assume that one third of the triples satisfy the requirements and divide c_t or $c_{p,t}$ in the above formulas by a factor of 3. If a `FILTER` contains an expression with the inequality operator (e.g. $?x! = 10$), we need to replace $\frac{1}{c_s}$ with $\frac{c_s-1}{c_s}$ because we do no longer select 1 value out of c_s different values but all except 1. The same consideration holds for c_o, $c_{p,s}$, and $c_{p,o}$.

Estimating Result Sizes for Joins

To estimate the sizes of join results, we need to distinguish between different shapes of joins: (1) star-shaped joins are characterized by multiple triple patterns joining on the same variable (e.g., `?s1 p1 ?o1 . ?s1 p2 ?o2`) and (2) path-shaped joins are characterized by multiple triple patterns that join on different variables (e.g., `?s1 p1 ?o1 . ?o1 p2 ?o2`).

To estimate the result size we use the cardinality estimation model proposed in [41]. The model proposes formulas for different types of joins. For example, for queries such as `SELECT ?y WHERE { ?x p1 ?y . ?x p2 ?o1 . FILTER(?o1=10) }` (star-shaped join) the cardinality is calculated as

$$c_{res} = \frac{\frac{c_{p2,t}}{c_{p2,o1}} \cdot c_{p1,t}}{max(c_{p2,x}, c_{p1,x})} \tag{3.5}$$

while for queries such as `SELECT ?x WHERE { ?x p1 ?y . ?y p2 ?o1 . FILTER(?o1=10) }` (path-shaped join) the cardinality is calculated as

$$c_{res} = \frac{\frac{c_{p2,t}}{c_{p2,o1}} \cdot c_{p1,t}}{max(c_{p1,y}, c_{p2,y})} \tag{3.6}$$

Estimating Result Sizes for Grouping and Aggregation

The number of results after evaluating grouping and aggregation depends on the size of the input. Given a list of attributes in the `GROUP BY` clause, the upper bound of tuples after grouping is the size of the input. Hence, in such a non-restrictive case we have $c_{res} = c_{in}$.

If the `GROUP BY` clause contains only a subset $(?x_1, \ldots ?x_n)$ of the variables contained in the query, then c_{res} (or more specifically c_{AGG}) is bound by the product of the variables' distinct bindings $\prod_{i=1}^{n} distinct(?x_i)$.

When solution reducers are present in the query, such as FILTER statements and/or triples with literals, which are connected to grouping variables through joins, we assume that the number of distinct values is reduced proportionally:

$$distinct(?x) = \frac{c_{p_x,x}}{c_{p_y,y} \cdot N} \tag{3.7}$$

where $c_{p_x,x}$ is the number of distinct bindings for variable *?x*, $c_{p_y,y}$ number of distinct bindings for variable ?y which is connected to *?x* through star-shaped or path-shaped joins, and N is the reduction factor, which is equal to 1 in case of the solution reducer with equality, 1/3 in case of solution reducer with inequality and $\frac{c_{p_y,y}-1}{c_{p_y,y}}$ in case of the solution reducer with negation (same as described in Estimating Result Sizes for Basic Triple Patterns).

For example for a query of type `SELECT COUNT(?x) ?y WHERE { ?x p1 ?o1 . ?x p2 ?y . FILTER(?o1=10) } GROUP BY ?y` the number of distinct values is equal to

$$distinct(?y) = \frac{c_{p2,y}}{c_{p1,o1} \cdot N} \tag{3.8}$$

where N is the number of distinct values specified by FILTER and for the equation (?o1=10) is equal to 1.

6 Evaluation

In this section, we present the results of evaluating the strategies presented in this chapter. Our solution uses the .NET Framework 4.0 and dotNetRDF (`http://dotnetrdf.org/`) to implement a mediator that accepts queries, optimizes their execution using the proposed strategies (SemiJoin, PartialAgg, and MedJoin), and sends subqueries to the SPARQL endpoints, which are using Virtuoso as local triple store.

6.1 Experimental Setup

We evaluate our strategies based on a standard benchmark originally designed to measure the performance of aggregate queries in relational database systems: the Star Schema Benchmark (SSB) [66]. This benchmark is well-known in the database community and was chosen for its simple design (refined decision support benchmark TPC-H [67]) and its well-defined testbed.

RDF Dataset. The data in SSB is generated as relational data. We used different scale factors (1 to 5 – 6M to 30M observations) to generated multiple datasets of different sizes. We translated the datasets into RDF using a vocabulary that strongly resembles the SSB tabular structure. For example, a lineorder tuple is represented as a star-shaped set of triples where the subject (URI) is linked via a property (e.g., `rdfh:lo_orderdate`) to a an object (e.g., `rdfh:lo_orderdate_19931201`) which in turn can be subject of another star-shaped graph. Values such as quantity and discount are connected to lineorder entities as literals. A simplified schema of the RDF structure is illustrated in Figure 3.1 while details about the generated datasets are given in

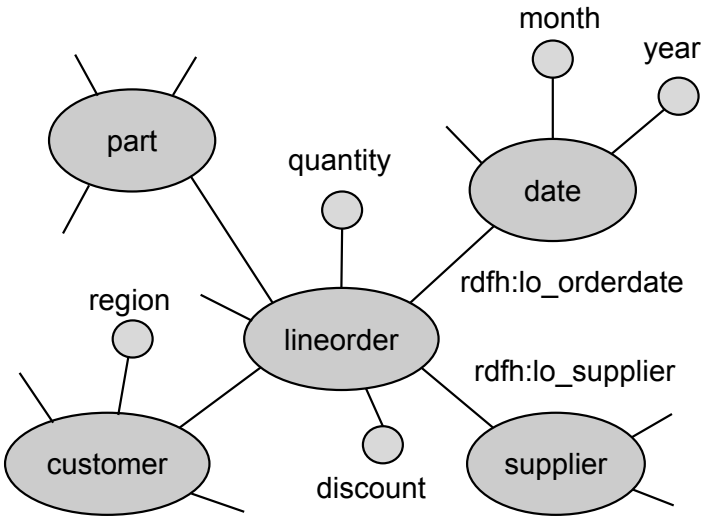


Fig. 3.1: Simplified Description of the SSB dataset

Table 3.1. Converted datasets contain 110,5M (scale factor 1) to 547,5M (scale factor 5) triples.

	LineOrder		Parts		Customers		Suppliers		Dates	
	Tuples	Triples	Tuples	Triples	Tuples	Triples	Tuples	Triples	Tuples	Triples
Scale Factor 1	6M	108M	220,000	2.2M	30,000	270,000	2000	16,000	2556	48564
Scale Factor 2	12M	216M	400,000	4M	60,000	540,000	4000	32,000	2556	48564
Scale Factor 3	18M	324M	400,000	4M	90,000	810,000	6000	48,000	2556	48564
Scale Factor 4	24M	432M	600,000	6M	120,000	1.08M	8000	64,000	2556	48564
Scale Factor 5	30M	540M	600,000	6M	150,000	1.35M	10000	80,000	2556	48564

Table 3.1: SSB Data Size

Queries. SSB defines 13 queries. They represent 4 "prototypical" queries with different selectivity factors. A brief description of the queries is given in Table 5.2. We converted all 13 queries into SPARQL and used the `SERVICE` keyword to query federated endpoints.

Configuration. To test the queries in a federation of SPARQL endpoints, we partitioned the datasets as follows:

- To simulate two endpoints (one endpoint containing main observation data and one `SERVICE` endpoint containing supporting data), we created two partitions: partition 1 (lineorders, parts, customers, and suppliers) and partition 2 (dates).
- To simulate three endpoints (two `SERVICE` endpoints containing sup-

Query Prototypes	Query No	Query Parameters for Various Selectivities
Prototype 1. Amount of revenue increase that would have resulted from eliminating certain company-wide discounts.	Q1.1	Discounts 1, 2, and 3 for quantities less than 25 shipped in 1993.
	Q1.2	Discounts 1, 2, and 3 for quantities less than 25 shipped in 01/1993.
	Q1.3	Discounts 5, 6, and 7 for quantities less than 35 shipped in week 6 of 1993.
Prototype 2. Revenue for some product classes, for suppliers in a certain region, grouped by more restrictive product classes and all years.	Q2.1	Revenue for 'MFGR#12' category, for suppliers in America
	Q2.2	Revenue for brands 'MFGR#2221' to 'MFGR#2228', for suppliers in Asia
	Q2.3	Revenue for brand 'MFGR#2239' for suppliers in Europe
Prototype 3. Revenue for some product classes, for suppliers in a certain region, grouped by more restrictive product classes and all years.	Q3.1	For Asian suppliers and customers in 1992-1997
	Q3.2	For US suppliers and customers in 1992-1997
	Q3.3	For specific UK cities suppliers and customers in 1992-1997
	Q3.4	For specific UK cities suppliers and customers in 12/1997
Prototype 4. Aggregate profit, measured by subtracting revenue from supply cost.	Q4.1	For American suppliers and customers for manufacturers 'MFGR#1' or 'MFGR#2' in 1992
	Q4.2	For American suppliers and customers for manufacturers 'MFGR#1' or 'MFGR#2' in 1997-1998
	Q4.3	For American customers and US suppliers for category 'MFGR#14' in 1997-1998

Table 3.2: SSB Queries

porting data), we created three partitions: partition 1 (lineorders, parts, customers), partition 2 (dates), and partition 3 (suppliers).

- To simulate four endpoints (three `SERVICE` endpoints containing supporting data), we created four partitions: partition 1 (lineorders, parts), partition 2 (dates), partition 3 (suppliers), and partition 4 (customers).

All the queries and the datasets used for the experiments are available at `http://extbi.cs.aau.dk/coda`.

We used four different machines for our experiments depending on the configuration. We used the most powerful machine (CPU Intel(R) Core(TM) i7-950, RAM 24 GB, HDD 1.5TB RAID5, 1TB SATA, 600GB SAS RAID0) for partition 1. We used three identical machines (CPU AMD(R) Opteron(TM) 285 2.6GHz, RAM 8GB, HDD 80GB) for serving data of partitions 2 to 4. 64-bit Ubuntu 14.04 LTS operating system was installed on all computers. As a mediator, we used a virtual machine with one dedicated core of Xeon E3-1240V2 3.4 GHz (2 threads), 10 GB RAM, 100 GB HDD, and 64-bit Windows Server 2008 Service Pack 1 as operating system. All machines were located on the same LAN. All benchmark queries were executed 5 times following a single warm-up run. During this warm-up run, all statistics and system measurements were obtained, stored in the system, and later used for the subsequent query executions. Statistics were gathered with the help of COUNT queries. Statistics collection took between 18 (scale factor 1) to 129 (scale factor 5) seconds. The execution time for each query is measured on the mediator from the time the query is received from a user till the time

the complete results are reported back. We used a timeout of 1 hour for the experiments.

6.2 Experimental Results

As discussed in Section 1, we initially experimented with three systems (Virtuoso, Sesame, and Jena Fuseki). Sesame is always trying to download all triples that match the patterns defined in the `SERVICE` subquery from the remote endpoint and is timing out even for small datasets. Jena Fuseki and Virtuoso are using the same strategy to evaluate `SERVICE` subqueries with grouping and aggregation. We chose Virtuoso v07.10.3207 as representative for this strategy in our experiments and include results for a native Virtuoso setup, in which Virtuoso is optimizing the distributed execution of the aggregate query.

In our first line of experiments, we measured the runtime for the benchmark queries in the configuration with one SPARQL endpoint. For the SemiJoin strategy, due to issues with large numbers of bindings in the `VALUES` clause in existing endpoints [19], we often have to partition the set of bindings that we aim to pass in a `VALUES` statement into smaller partitions and send a separate messages for each of the partitions.

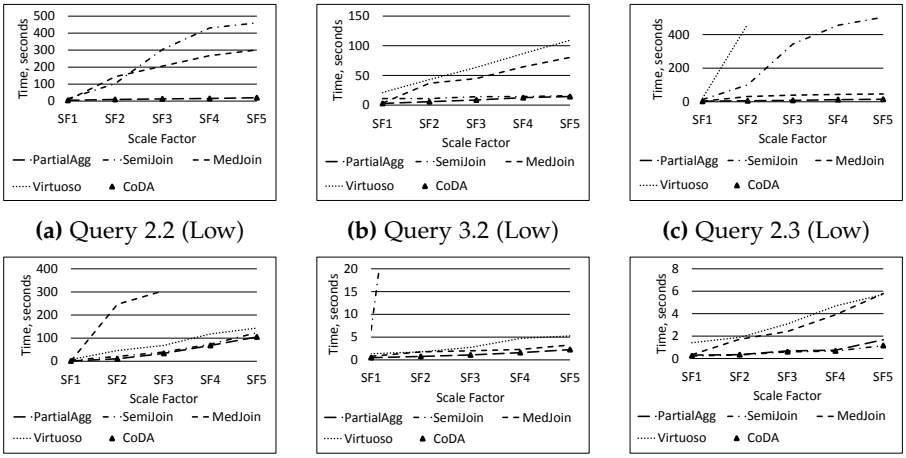
	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3	Q3.1	Q3.2	Q3.3	Q3.4	Q4.1	Q4.2	Q4.3
Scale Factor 1													
Virtuoso	T/O	T/O	760	500,3	107,8	21,3	215,8	21,2	1,4	1,4	863	969	7,3
SemiJoin	**1,3**	**0,2**	**0,1**	12,7	13,5	12,6	**14,1**	11,0	6,5	**0,2**	4,8	**8,1**	4,5
PartialAgg	1,6	1,4	0,8	**9,4**	**4,5**	3	17,5	**2,8**	**0,5**	0,3	**4**	18,5	**1,0**
MedJoin	249,5	213,4	82,9	11	5,2	**2,9**	98,9	3,4	0,8	0,3	26,4	32	1,1
CoDA	**1,3**	**0,2**	**0,1**	**9,4**	**4,5**	**2,9**	**14,1**	**2,8**	**0,5**	**0,2**	**4**	**8,1**	**1,0**
Scale Factor 2													
Virtuoso	T/O	T/O	T/O	950,9	T/O	462,9	992,2	42,9	1,8	1,9	T/O	1054	46,5
SemiJoin	**3,6**	**0,9**	**0,5**	25,7	102,8	101	**15,4**	11,1	89,7	**0,32**	**30,6**	**35,5**	20,6
PartialAgg	17,1	16,5	7,3	**16,2**	**9,5**	**5,9**	18,4	**5,8**	**0,8**	0,34	77,3	37,4	**10,5**
MedJoin	T/O	T/O	T/O	T/O	143,7	31,5	612,7	36,7	1,8	1,7	T/O	T/O	246,7
CoDA	**3,6**	**0,9**	**0,5**	**16,2**	**9,5**	**5,9**	**15,4**	**5,8**	**0,8**	0,33	**30,6**	**35,5**	**10,5**
Scale Factor 3													
Virtuoso	T/O	T/O	T/O	1465	T/O	T/O	T/O	63,5	2,8	3,1	T/O	T/O	68,5
SemiJoin	46,3	**5,4**	**2,2**	330,7	303,4	344,1	**20,2**	14,2	250,7	**0,6**	**45,4**	**105,3**	39,8
PartialAgg	**18,4**	18,8	8,3	**29,5**	**13,2**	**8,2**	23,2	**8,6**	**1,1**	0,7	217,4	606	**33,9**
MedJoin	T/O	T/O	T/O	T/O	205,7	39,5	1312	44,8	2	2,4	T/O	T/O	305,3
CoDA	**18,4**	**5,4**	**2,2**	**29,5**	**13,2**	**8,2**	**20,2**	**8,6**	**1,1**	**0,6**	**45,4**	**105,3**	**33,9**
Scale Factor 4													
Virtuoso	T/O	T/O	T/O	T/O	T/O	T/O	T/O	86,9	4,7	4,7	T/O	T/O	118,4
SemiJoin	64,2	**6.9**	**2,4**	368,5	430,3	455,4	**23,7**	14,5	275,6	**0,7**	**54,2**	**116,2**	73,5
PartialAgg	**33,9**	27,6	9,8	**146,2**	**15,2**	**12,9**	27,2	**12,5**	**1,6**	0,8	980,8	1017	**68,3**
MedJoin	T/O	T/O	T/O	T/O	267,5	43,6	T/O	64,5	2,3	3,9	T/O	T/O	T/O
CoDA	**33,9**	**6.9**	**2,4**	**146,2**	**15,2**	**12,9**	**23,7**	**12,5**	**1,6**	**0,7**	**54,2**	**116,2**	**68,3**
Scale Factor 5													
Virtuoso	T/O	T/O	T/O	T/O	T/O	T/O	T/O	109,2	5,3	5,7	T/O	T/O	143,4
SemiJoin	77,7	**8,4**	**2,9**	453,4	460,3	503,6	**60,9**	15,8	352,9	**1,2**	**59,2**	**126,8**	123,6
PartialAgg	**37,7**	29,2	18,4	**249,5**	**19,8**	**14,9**	78,5	**14,4**	**2,2**	1,7	1565	1577	**105,1**
MedJoin	T/O	T/O	T/O	T/O	301,2	46,3	T/O	80,4	3,3	5,8	T/O	T/O	T/O
CoDA	**37,7**	**8,4**	**2,9**	**249,5**	**19,8**	**14,9**	**60,9**	**14,4**	**2,2**	**1,2**	**59,2**	**126,8**	**105,1**

Table 3.3: Benchmark Results For Scale Factor 1 to 5, in seconds

Table 3.3 shows the results for scale factors 1 to 5. CoDA clearly chooses the best strategy for all queries. For scale factor 1, the CoDA algorithm selected the SemiJoin strategy for queries with highly selective subqueries

(where the number of intermediate subquery results are low) (Q1.1, Q1.2, Q1.3, Q3.1, Q3.4, and Q4.2), the MedJoin strategy for queries with high selectivity (Q2.3), and the PartialAgg strategy for the rest.

CoDA scales well with the increase in the number of triples as the results for scale factors 2 to 5 in Table 3.3 show. Due to the increased number of triples to process, the strategy for Query 2.3 changes from MedJoin to PatrialAgg. CoDA also changed the strategies for queries 1.1 and 4.1 due to different estimations of C_C and C_P for various scale factors. In general, CoDA chooses the best strategy for all queries (the difference between the CoDA approach and the best approach for query Q3.4 in scale factor 2 is due to the overhead of optimization, which is only 14 ms).



(a) Query 2.2 (Low) **(b)** Query 3.2 (Low) **(c)** Query 2.3 (Low)

(d) Query 4.3 (High) **(e)** Query 3.3 (High) **(f)** Query 3.4 (High)

Fig. 3.2: Execution Times for Queries with Low and High Selectivity, One Endpoint

Figure 3.2 shows the execution times for several queries with high selectivity (Q4.3, Q3.3, Q3.4) and low selectivity (Q2.2, Q3.2, Q2.3) for different strategies and scale factors – due to timeouts in execution, some lines end earlier than others. MedJoin and native Virtuoso do not scale well and some queries time out while SemiJoin and PartialAgg return answers for all the queries. This can be explained by the internal logic behind the strategies. For example, Virtuoso sends SPARQL requests for every aggregated observation, while MedJoin needs to transfer much data to the mediator. Due to the result size restrictions (the maximum result set size for Virtuoso is 1,048,576), the system downloads all data in chunks but still times out. In contrast, SemiJoin and PartialAgg transfer only necessary data and are thus reducing the communication costs.

We also evaluated the influence of the number of endpoints. For this pur-

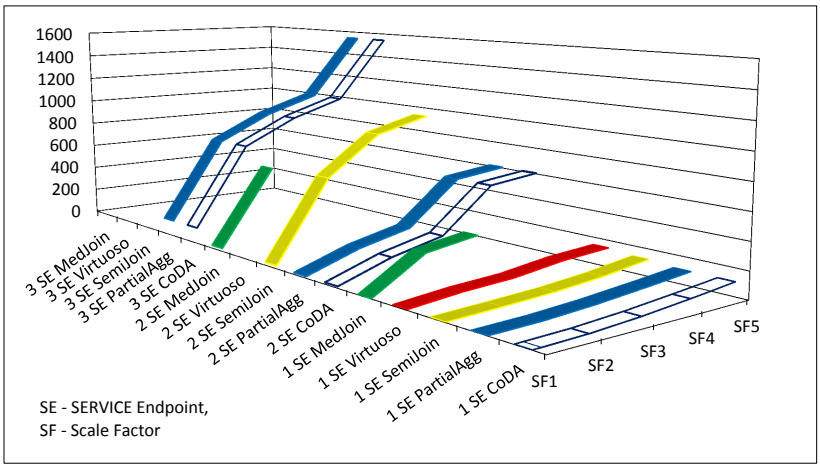


Fig. 3.3: Execution of Query 4.3 over Several Endpoints

pose, we chose an example query from our workload (Q4.3) that is complex enough to be rewritten into a query with up to three `SERVICE` endpoints and selective enough not to require all triples for the calculation (Figure 3.3). Going up to three endpoints, only the PartialAgg strategy was able to answer the query. With data coming from two or three endpoints, the number of values that needs to be passed in the SemiJoin strategy increases and system performance quickly degrades (yellow lines in Figure 3.3). With the partition of the dataset into more endpoints, MedJoin also needs to load much more data into the mediator site to answer the query and for the scale factors 3 to 5 this leads to timeouts (green lines in Figure 3.3). The same reason (the need to send more requests to answer the query) leads to the timeout in the Virtuoso strategy (red lines) for queries with more than one `SERVICE` endpoint. Therefore, the obvious choice of the CoDA strategy is PartialAgg (blue lines) in these cases.

In summary, the experimental results show that CoDA is able to select the best strategy and thus executes all queries for RDF data of all tested data sizes.

7 Conclusions and Future Work

Motivated by the increasing availability of RDF data over SPARQL endpoints, the new powerful aggregation functionality in SPARQL 1.1, and the desire to perform ad-hoc analytical queries, this chapter investigated the problem of efficiently processing aggregate queries in a federation of SPARQL endpoints.

More precisely, the chapter proposed the Mediator Join, SemiJoin, and

Partial Aggregation query processing strategies for this scenario. The chapter also proposed a cost model, and techniques for estimating constants and result sizes for triple patterns, joins, grouping and aggregation, and the combination of these with the processing strategies into the Cost-based Optimizer for Distributed Aggregate queries (CoDA) approach for aggregate SPARQL queries over endpoint federations. The comprehensive experimental evaluation, based on an RDF version of the widely used Star Schema Benchmark, showed that CoDA is efficient and scalable, able to pick the best query processing plan in different situations, and significantly outperforms current state-of-the art triple stores.

Interesting directions for future work include using more complex statistics with precomputed join result sizes and correlation information to better estimate cardinalities, optimizing the execution of more complex queries (e.g., with optional patterns or complex aggregation functions), and investigating the influence of ontological constraints and inference/reasoning in the context of federated aggregate SPARQL queries.

Chapter 4

Efficient Support of Analytical SPARQL Queries in Federated Systems

Dilshod Ibragimov, Torben Bach Pedersen, Katja Hose and Esteban Zimányi

The paper is to be submitted to a conference.

The layout of the paper has been revised.

Abstract

*As more and more data is becoming accessible on the Web via SPARQL endpoints, we can often find related data at multiple endpoints. Hence, diverse query processing strategies have been developed to overcome challenges such as links between sources, heterogeneity of the data sources, the need to support implicit (derived) information and reasoning, etc. However, not many of these techniques have been designed to support analytical SPARQL queries involving grouping and aggregation, which are an integral part of online analytical processing (OLAP) systems and enable interesting insights and analyses at a large scale. Hence, in this chapter we propose LITE (O**L**AP-style Analyt**I**cs in a Federa**T**ion of SPARQL **E**ndpoints), a federated system for computing aggregate SPARQL queries over a federation of SPARQL endpoints addressing the above mentioned challenges. In particular, LITE is able to integrate the diverse schemas of SPARQL endpoints and provide access to the data via OLAP-style hierarchies to enable uniform, efficient, and powerful analytics. The experimental evaluation shows that LITE significantly outperforms the state of the art.*

1 Introduction

More and more organizations publish data in RDF format and make it accessible via SPARQL endpoints. Yet, many users are interested not only in the data of a single endpoint but also in complex SPARQL queries that require combining partial results from multiple endpoints.

Consider, for example, a user who is interested in analyzing weather data from different countries as, for instance, provided by several public SPARQL endpoints[1,2,3]. Among other measures, each of these endpoints provides access to information about precipitation in a single country (Australia, Spain, USA). Let us assume that the user is interested in finding regions in each of these countries with the highest or lowest amount of precipitation in a given time range. Answering this query requires accessing each of these sources. Evaluating such queries does not only require efficient query execution strategies but also means to overcome the problem of data source heterogeneity, i.e., each source uses a different schema (ontology) to structure and describe the local data. Such heterogeneity makes it very difficult for a user to formulate a single query for complex information needs such as the example mentioned above.

Another typical scenario is when several endpoints contain statistics or census data of a single country. The user may be interested in the analysis of

[1] `http://lab.environment.data.gov.au/weather/sparql`

[2] `http://aemet.linkeddata.es/sparql`

[3] `http://sonicbanana.cs.wright.edu:8890/sparql`

data *across* these endpoints like aggregating data at the continent level using *external* hierarchies provided by other sources. The same scenario is valid for the previous example with data about precipitations. Suppose the SPARQL endpoints provide only information about the city where a sensor is located. Our user, however, might want to group the readings by districts, federal states, or continents, i.e., by information that is not present in these endpoints directly. As many publishers link their data to well-known datasets such as GeoNames (`http://www.geonames.org/`), we can use these links to retrieve missing information and hierarchy levels from other sources.

One of the interesting aspects that is neglected by most other approaches is entailment, i.e., deriving new information from data using RDF semantics. RDF Schema (RDFS) entailment patterns are particularly interesting for analytical queries since RDFS encodes the domain semantics. In particular, RDFS properties, such as *rdfs:subClassOf*, may define (implicit) hierarchies between entities and should therefore be considered to ensure complete answers to analytical queries.

To overcome the above mentioned challenges, we propose LITE, an approach that uses a mediated (global) schema comprising (the relevant parts of) all heterogeneous source schemas (local schemas) and that takes into account externally linked hierarchies and RDF entailment. A user of LITE does no longer have to be aware of the underlying federation of SPARQL endpoints but can conveniently formulate a query on the global schema and the system will automatically take care of all actions that are necessary to retrieve the final result.

In summary, the contributions of this chapter are:

- a native RDF/SPARQL-based approach for efficient support of analytical queries over federations of SPARQL endpoints,
- an extended vocabulary for specifying the mapping between the multidimensional mediated schema (global) and the local schemas of the sources, and
- an algorithm for rewriting SPARQL queries in a federation of endpoints that takes into account hierarchical information encoded in RDFS.

The remainder of this chapter is structured as follows. Section 2 discusses related work and Section 3 introduces the preliminaries. While Section 4 presents how LITE facilitates the integration of local schemas into a global schema, Section 5 presents how LITE rewrites and optimizes queries. Section 6 discusses our experimental results and Section 7 concludes the chapter.

2 Related Work

The problem of data integration has been studied extensively in the context of relational databases [71,88–91]. Such systems typically describe sources with views over a global mediated schema. Queries are then defined on the global schema and rewritten into queries over the source data. Most query rewriting approaches apply the Local-As-View paradigm, where the local schemas are expressed as views over the global schema, or the Global-As-View paradigm, where the global schema is expressed as a view over the local schemas. Techniques designed for relational systems cannot easily be adapted to RDF data. Whereas the schema in relational systems, for instance, is rather stable, RDF datasets and their schemas are much more dynamic and do not have the same structural properties as relations. Moreover, standard relational databases are limited to explicit data only while RDF systems support entailment.

Recently, some approaches for RDF data have been proposed. [68] proposes an RDF analytics framework supporting typical OLAP operations and analytical queries. Although hierarchies are supported by using a *nextLevel* property, the proposed framework does not offer techniques to handle complex or implicit hierarchies and is not intended for a federated setup. To facilitate query rewriting over multiple datasets, [92,93] adopt mappings between source and target ontologies using either Description Logic [93] or Ontology Alignment [92]. The SPARQL queries are rewritten using these predefined mappings. However, these approaches do not support aggregate queries, do not consider complex and implicit hierarchies, and do not account for inferred triples.

In [78], an algorithm for SPARQL query rewriting over a number of virtual SPARQL views is proposed; the approach removes redundant triple patterns originating from the same view and eliminates rewritings with empty results. An alternative technique to process SPARQL queries without generating rewritings is proposed in [94]. It ranks and aggregates relevant views into an aggregate view, which is used for query execution. Both approaches [78,94] focus on conjunctive queries and do not address the issues related to aggregation and dimensional hierarchies. [26] evaluates the performance gain of manually constructed RDF aggregate views that fully match the predefined set of queries. This work, however does not propose a novel algorithm for query rewriting or view selection. [69] presents cost-based optimization strategies for aggregate queries over linked data that are suitable for a single endpoint and does not support the integration of similar data from various endpoints. Besides, implicit triples were not considered. To speed-up query processing on SPARQL endpoints, [95] proposes a cost model and techniques for selecting a set of aggregate RDF views to materialize and an algorithm for rewriting user queries given a set of materialized views. However,

the approach in [95] is designed for single endpoints and does not address issues related to processing queries in a federation of endpoints.

The literature also proposes several approaches for querying federated RDF sources. FedX [18] uses SPARQL ASK queries for source selection and implements bound joins to reduce the number of requests between sources. ANAPSID [55] uses a catalog of endpoint descriptions to decompose a user query into subqueries for the execution by separate endpoints and pipelines the result to other operators in the query execution tree. SI-HJoin [58] uses a hash join implementation to enable pipelining in combination with a lightweight cost-model. [96] is designed for federations using triple pattern fragments and optimizes query execution in the presence of replicated data by selecting useful fragments and minimizing the amount of transferred data. DARQ [97] uses metadata (service descriptions) to identify relevant sources for a given query and HiBISCus [98] addresses the problem of source selection by discarding sources that are not pertinent to the final result computation. All these approaches are designed for conjunctive queries and are not directly applicable in the context of analytical queries.

In summary, most state-of-the-art approaches for federated SPARQL query processing are designed with a focus on conjunctive queries or do not offer support or optimizations for multidimensional analytical queries. In contrast, this chapter proposes a novel approach for executing analytical queries in a federation of SPARQL endpoints over dynamic, graph-structured multidimensional data taking into account RDF specifics.

3 RDF Graph and Queries

The notation we use in this chapter follows [99] and is based on three disjoint sets: blank nodes B, literals L, and IRIs I (together BLI). An RDF triple $(s, p, o) \in BI \times I \times BLI$ connects a subject s through property p to object o. A subject usually denotes an *entity* or a *class*, while an object denotes an *entity*, a *class* or a *literal value*. An RDF dataset (G) consists of a finite set of triples and is often represented as a graph. Figure 4.1 shows an example RDF graph describing precipitation observations in Spain (extracted from `http://aemet.linkeddata.es/sparql`) with links to DBpedia.

Definition (RDF Graph) An RDF graph is denoted as $G = \{N, E, \Theta\}$, where $N = N_E \cup N_C \cup N_L$ is the set of nodes corresponding to subjects and objects in a set of RDF triples (with N_E, N_C, and N_L representing sets of entity nodes, class nodes, and literal nodes); $E \subseteq N \times N \times \Theta$ is the set of directed edges representing triples; and Θ is the set of corresponding edge labels representing the triples' predicates.

Queries are based on graph patterns that are matched against G. A Basic Graph Pattern (BGP) consists of a set of triple patterns of the form $(IV) \times$

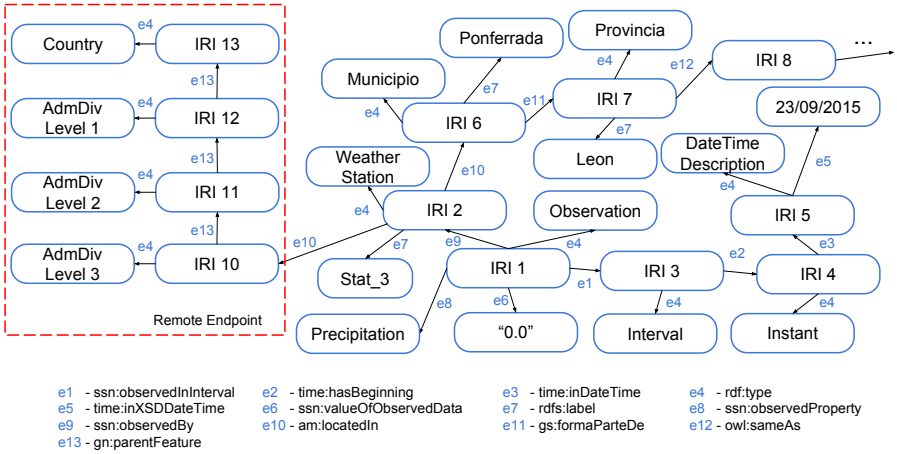


Fig. 4.1: Spanish precipitation observations

$(IV) \times (LIV)$, where V ($V \cap BLI = \emptyset$) is a set of query variables.

Definition (BGP Query graph) A BGP query is denoted as $Q = \{N^Q, E^Q, \Theta^Q\}$, where N^Q is a set of nodes and variables; E^Q is a set of edges and variables; and Θ^Q is the corresponding set of edge labels.

Most SPARQL aggregate queries use the pattern *SELECT* RD *WHERE* GP *GROUP BY* GRP, where GRP defines a set of *grouping variables* and GP a BGP optionally using functions, such as assignments (e.g., *BIND*) and constraints (e.g., *FILTER*). RD is the set of *result description variables* corresponding to a subset of variables in the graph pattern GP and *aggregation variables* with corresponding aggregate functions.

In this chapter, we focus on analytical queries on RDF data for aggregating measurable attributes (measures) of a set of observations (facts) according to relevant criteria (dimensions). The basic graph pattern of such queries usually has a special *rooted* pattern [68].

Definition (Rooted Query) A BGP of query q is rooted in node $n \in N$ iff there exists a connected path from any node $x \in N$ to node n.

4 Data Integration in LITE

In this section we first describe how LITE models global and local schemas as graphs. We then proceed to describe how we define mappings between these schemas.

4.1 Modeling Source and Target Schemas

To represent schemas of the *local* sources we use an RDF schema graph that *highlights* the structure of the data available to the analyst, i.e., a *footprint* of the available data.

Definition (RDF Schema Graph) The schema graph of an RDF dataset G is denoted as $G^S = \{N^S, E^S, \Theta^S\}$, where $N^S = N_A \cup N_C$ is a union of summary nodes N_A and class nodes in the ontology N_C ; $E^S \subseteq N^S \times N^S \times \Theta^S$ is a set of (directed) edges in the RDF schema graph ($E^S \subseteq E; E \in G$); and Θ^S is a set of corresponding edge labels. Each summary node $n_A \in N_A$ represents an aggregation of all entity nodes $n_E \in N_E$ that are instances of a specific class $c \in N_C$ or literal nodes $n_L \in N_L$.

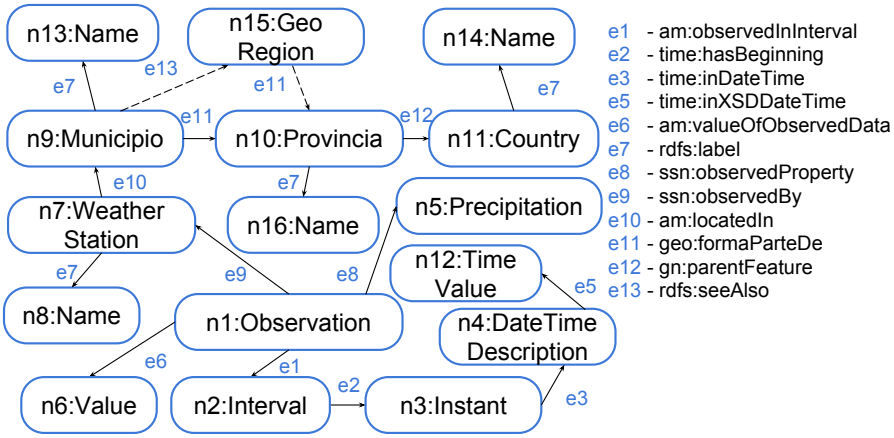


Fig. 4.2: RDF Schema graph for Spain

We use the notion of *summary nodes* to represent the repetitive nodes in a dataset that have the same class but different IRIs, like observations, weather stations, etc. We use the summary nodes during the query rewriting phase and replace these nodes with variables if necessary. The schema graph of the RDF data graph given in Figure 4.1 (without data from the remote endpoint) is shown in Figure 4.2.

```
ls:node1 a am:Observation .
ls:node1 am:observedInInterval ls:node2 .
ls:node1 ssn:observedBy ls:node7 .
ls:node1 am:valueOfObservedData ls:node6 .
ls:node2 a time:Interval .
ls:node2 time:hasBeginning ls:node3 .
ls:node3 a time:Instant .
ls:node3 time:inDateTime ls:node4 .
ls:node4 a time:DateTimeDescription .
ls:node4 time:inXSDDateTime ls:node12 .
ls:node7 a am:WeatherStation .
ls:node7 rdfs:label ls:node8 .
```

```
ls:node7 am:locatedIn ls:node9 .
ls:node9 a geo:Municipio .
ls:node9 rdfs:label ls:node13 .
ls:node9 geo:formaParteDe ls:node10 .
ls:node10 a geo:Provincia .
ls:node10 rdfs:label ls:node16 .
```

Listing 4.1: Excerpt of local schema

Multidimensional global schema: The *global* schema, on the other hand, can be built using powerful multidimensional vocabularies like QB4OLAP [25], and (to a lesser extent) QB [30] or more general approaches like AnS [68]. While the global schema should in general (at least) be able to represent the structure of the virtually integrated data, LITE with QB4OLAP offers much more expressive power by natively supporting well-known, intutive data analytics concepts such as dimensions with hierarchies of levels (for example, a hierarchy generalizing the administrative division of countries) and built-in measures with associated (automatic) aggregation functions (for example total precipitation over a group of observations). We note that the RDF schema graph is not appropriate for defining the global schema since it does not allow us to specify the dimensions, complex hierarchies or measures as needed by analysts.

In general, analysts prefer to work with highly structured data. However, the structure of the local sources may be less evident. Thus we should map the local schemas to the global schema to allow analysts to write structured, multidimensional (OLAP-like) queries using the global schema that will then be automatically translated into queries over local graph-like data with a less explicit data structure.

4.2 Mapping Model

We now outline the mapping model that map the local and global schemas in LITE. In a sense, we are mapping a multidimensional data schema to a schema that represents graph-like data. However, since both local and global schemas are encoded as RDF, both local and global schemas can be represented as graphs. Therefore, we depict the global schema as a graph where nodes represent the concepts specified in the multidimensional data model [25]. A root node, for example, represents observations that may be analyzed using other schema nodes reachable from the root node such as hierarchies of dimension levels and measures.

In our mapping model, we allow to specify schema fragments – the subgraphs that reflect some core concepts in schemas. For instance, in a schema, a fragment can be a subgraph that connects a node representing a measure to the root node. Another example of the schema fragment is a subgraph that represents a step in a hierarchy – a parent/child relationship between two levels. The schema fragments can easily model such concepts, unlike single

triples. For example, nodes *n1*, *n6*, *n5* and edges *e6*, *e8* in Figure 4.2 constitute a schema fragment related to the Precipitation measure, while nodes *n1*, *n7* and the edge *e9* constitute the Station level in the Geography dimension. The union of all fragments of the schema forms the complete schema. Then we map the fragments representing the same concepts in global and local schemas to each other.

Definition (RDF Schema Fragments) An RDF schema $S = \{N, E, \Theta\}$ (N - set of nodes, E - set of edges, Θ - set of edge labels) can conceptually be partitioned into a set of fragments $F = \{F_1, F_2, ..., F_m\}$ where $F_k, k = 1, ..., m$ is specified by $(N_k^{in} \cup N_k^{ex}, E_k, \Theta_k)$ such that
1. $[N_1^{in} \cup N_1^{ex}, ..., N_m^{in} \cup N_m^{ex}]$ is a disjoint partitioning of N, i.e., $N_i^{in} \cap N_j^{in} = \emptyset, i, j \in \{1, \dots m\}, i \neq j$, and $[\cup_{k=1,...,m} N_k^{in}, \cup_{k=1,...,m} N_k^{ex}] = N$;
2. A node n $\in N_k^{ex}$ iff the node n resides in several fragments, i.e. n $\in F_i, n \in F_j, i \neq j$;
3. Nodes in N_k^{ex} are called external nodes and nodes in N_k^{in} are called internal nodes of F_k;
4. $E_k \subseteq N_k^{in} \times N_k^{in} \times \Theta$;
5. Θ_k is a set of edge labels in F_k.

In the example schema fragment related to the measure, node *n1* is an external node (also present in other fragments), while node *n6* is internal.
BGP Query fragments Similarly with Definition 4.2, a BGP query graph consists of a set of BGP query fragments where variables may appear in any of the subject, predicate, or object positions within the fragments.

Our mapping model also links the graph nodes that have the same meaning in both global and local schemas. Since the global schema is built based on the local schemas, measures and hierarchy levels present in the global schema should contain their counterpart nodes in the local schemas. Hence, in LITE we link these nodes. For example, if the node represents a hierarchy level in the global schema, it can be linked to a node representing the same hierarchy level in the local schema. LITE also allows extra hierarchies to be added as will be explained later.

SPIN Extension: To map schema fragments and link nodes in global and local schemas, we extend the SPARQL Inferencing Notation (SPIN) Syntax vocabulary [20] – an RDF syntax to represent SPARQL queries as RDF triples. The SPIN SPARQL Syntax can represent SPARQL queries and thus can encode graph patterns of arbitrary complexity. We adapt SPIN to represent RDF schema graphs. For example, to represent the constituents of triples, we use such SPIN properties as *sp:subject*, *sp:predicate*, and *sp:object*. The set of triples is represented as a list (*rdf:List*) and is defined by the *sp:elements* property. Also, the vocabulary can encode optional triples (*sp:Optional*), combination of alternative triples (*sp:Union*), negation of triples (*sp:Minus* and *sp:NotExists*), etc.

Additionally, SPIN can encode fragments that reside on a remote endpoint. To do so, SPIN uses the instances of *sp:Service*. The property *sp:serviceURI* points to the URI where the data is located. This helps to map global schema hierarchies to hierarchies that are external for local data sources. The set of graph patterns related to the external hierarchies is represented with the property *sp:elements*. Listing 4.2 shows an example of defining an external dataset with *sp:Service* property.

```
1   ex:gsmt4 fs:pattern [sp:subject gs:node1; sp:predicate ex:station; sp:
    object gs:node7 ] .
2   ex:gsmt5 fs:pattern [sp:subject gs:node7; sp:predicate qb4o:inLevel; sp:
    object ex:Station ] .
3   ex:gsmt6 fs:pattern [sp:subject gs:node7; sp:predicate skos:broader; sp:
    object gs:node17 ] .
4   ex:gsmt7 fs:pattern [sp:subject gs:node17; sp:predicate qb4o:inLevel; sp
    :object ex:AdmDivLevel2 ] .
5   ex:glist3 sp:elements ex:gsmt4, ex:gsmt5 . ex:glist4 sp:elements ex:
    gsmt6, ex:gsmt7 .
6   ex:lsmt5 fs:pattern [sp:subject ls:node1; sp:predicate ssn:observedBy;
    sp:object ls:node7 ] .
7   ex:lsmt6 fs:pattern [sp:subject ls:node7; sp:predicate owl:locatedIn; sp
    :object ls:node9 ] .
8   ex:lsmt7 fs:pattern [sp:subject ls:node9; sp:predicate owl:sameAs; sp:
    object gn:node12 ] .
9   ex:lsmt8 fs:pattern [sp:subject gn:node12; sp:predicate gn:name; sp:
    object gn:node16 ] .
10  ex:llist3 sp:elements ex:lsmt5 . ex:llist4 sp:elements ex:lsmt6, ex:
    lsmt7 .
11  ex:glist3 fs:schemaMatch ex:llist3 . gs:node7 fs:sameConcept ls:node7 .
12  ex:glist4 fs:schemaMatch ex:llist4 . gs:node17 fs:sameConcept ls:node13
    .
13  ex:gnsrv a sp:Service ; sp:serviceURI <http://lod2.openlinksw.com/sparql
    > ; sp:elements ex:lsmt8 .
```

Listing 4.2: Mapping global and local schemas

In a typical scenario, a graph pattern in SPIN is stored as a (tree) structure of blank nodes (for example, a triple pattern that has exactly one value for the properties sp:subject, sp:predicate and sp:object is represented as a blank node). To map schema fragments, we encode these fragments as a list of triples and associate it with the fragments (a list of triples) from other schemas. To benefit from using SPIN for mapping the schema fragments, we need to introduce properties to properly identify each triple in the list and correctly map these lists of triples from different schemas. Therefore we introduce the property names whose meanings are intuitive:

`pattern`, `schemaMatch`, `sameConcept`

The property *pattern* is used to identify each triple in the schema graph. Then, these triples are used to form a list of schema fragment triples. The property *schemaMatch* is used to map the triples that represent the same fragments in a local and global schemas. The property *sameConcept*, on the other hand, is used to link the nodes that have the same meaning in both schemas.

Consider the excerpt from the mappings produced for the global and local schemas (Listing 4.2) that maps schema fragments representing the parent-

child relationship between the observation and the first level of the Administrative Division hierarchy in both schemas. The *fs:pattern* predicate is used to construct triples that represent one triple of a graph (lines 1-4 and 6-9). The subgraphs that constitute the fragment in the schema meaningful for the data analysis is represented as a list of these triples with the help of *sp:elements* property (lines 5 and 10). Then the schema fragments that represent the same concept in different schemas are mapped with the property *fs:schemaMatch*, while the graph nodes that denote the same meaning in both schemas (the station and the location of the sensors) are mapped with the property *fs:sameConcept* (line 11-12). The property *sp:Service* represents a remote endpoint that holds data related to externally defined hierarchies (line 13). Graphically, this process can be represented as shown in Figure 4.3 where the mapped parts of both schemas are highlighted. To the left, a global hierarchy generalizing the administrative divisions of countries is added, based on an external ontology. We see how some of the lower levels (in red) map to the local source, while the new upper levels give the analysts new opportunities to aggregate data beyond what the local sources can support.

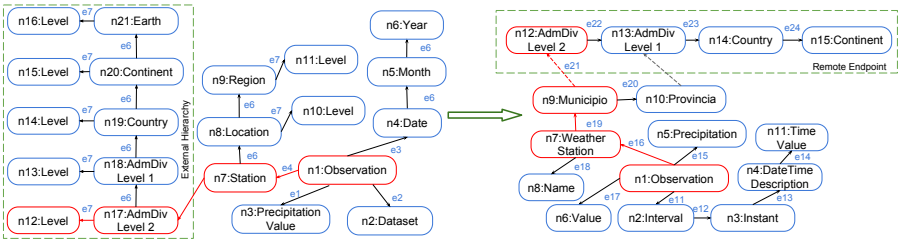


Fig. 4.3: Mapping global and local schemas

5 Query Rewriting and Optimization

In this section we describe our query rewriting algorithm to rewrite a globally defined query to corresponding queries over local datasets. Then we specify the techniques we used to optimize rewritten queries. During query rewriting, we also apply RDFS rules to account for implicitly defined hierarchies in local datasets.

5.1 Query Rewriting

With the schemas and the mapping defined, the system can process the queries written by the users. The system rewrites the queries defined over the global schema to be executed over the local data sources. The rewritten query is produced by replacing the query graph patterns defined over the global schema with the graph patterns defined over the local schemas

and rewriting the aggregate function. In our approach, we focus on analytical RDF queries over hierarchical multidimensional data that are linked to remote data sources. We also expect that the schemas of the sources can be divided into a set of fragments representing different concepts in a data model like a measure or a step in a hierarchy. At the same time we suppose that the global and local schemas contain nodes representing the same concepts in both schemas. The query rewriting algorithm is described in Algorithm 6.

First, we need to identify the global schema subgraph that corresponds to the user query by matching the BGP of the user query against the RDF schema graph (line 1).

Definition (BGP query match against the RDF schema graph) A graph S' is a match of query $Q\{N^Q, E^Q, \Theta^Q\}$ over RDF schema graph $S\{N, E, \Theta\}$ ($S' = Q(S)$) if and only if S' is a subgraph of S and exists a mapping function f that:
– maps a variable in N^Q to any node in N;
– maps a node $n^q \in N^Q$ to a node $n^s \in N$ if nodes n^q and n^s have the same URI;
– maps an edge $e^q \in E^Q$ to an edge $e^s \in E$ such that the label of the edge e^q in Θ^Q is equal to the label of the edge e^s in Θ;
– maps a variable in E^Q to any edge in E.

Then, from all possible schema fragments, we identify the subset of the fragments such that these fragments, when combined, constitute the subgraph produced during the previous step (line 3). Using these schema fragments and the mapping among the schema fragments, we build a schema of the local data source that corresponds to the user query (line 5). Then, based on globally available information, we optimize the local schema (the details of the optimization are discussed in Sec. 5.2). Additionally, using the global schema fragments, we identify the corresponding query fragments of the user query. Then, for each variable in the query fragment that is also a result description variable in the *SELECT* statement, we identify the corresponding node in the global schema and the same concept node in the local schema and replace the node in the local schema with the corresponding variable name (lines 10 - 13). After that, all the remaining summary nodes in the schema of the local data source are replaced with the newly generated variables so that the resulting graph patterns can be used in the local query. Then, the graph patterns in the *WHERE* clause of the global query are replaced with the graph patterns of the local query and the aggregate functions are rewritten (line 16) to account for the type of the function (algebraic or distributive). Next, the rewritten query is passed for further execution.

```
SELECT ?loc (SUM(?prec) as ?totalPrec)
WHERE {
  ?obs rdf:type qb:Observation .
```

Input: Global schema S_G, global query Q_G, mapping schema M
Output: Rewritten query Q_L

1 $S'_G = Q_G(S_G)$ -- a result of execution of query Q_G over global schema S_G ;
2 $F^G = [F^G_1, \ldots, F^G_k]$; $F^G \in M$ -- is a set of global schema fragments in M ;
3 From F^G find such subset $F'_G = [F^G_1, \ldots, F^G_m]$ that $F'_G \subseteq F^G$ and $\cup[F^G_1, \ldots, F^G_m] = S'$;
4 $F'_L = [F^L_1, \ldots, F^L_m]$ -- is a set of schema fragments for a local dataset from the mapping schema M where F^L_i is mapped to $F^G_i, i = [1, \ldots, m]$;
5 $S'_L = \cup F'_L$ -- is a schema of the local dataset corresponding to the query Q_G ;
6 $S'_L = ApplyOptimization(S'_L, M)$;
7 $F'_Q = [F^Q_1, \ldots, F^Q_m]$ -- is a set of query fragments from Q such that F^G_i is a schema match for F^Q_i for $i = [1, \ldots, m]$;
8 $D = \cup_{i=1,\ldots,m}(n^G_i, n^L_i)$ -- is a dictionary of same concept nodes for all fragments F'_G, F'_L from mapping M ;
9 $RD = vars_\pi(Q)$ -- is a set of projected (result description) variables in Q ;
10 **foreach** $F^Q_i \in F'_Q$, $F^S_i \in F'_S$ **do**
11 **if** $\exists var \in F^Q_i$ *and* $var \in RD$ *and* var *is mapped to* $n \in (N^i_i \cup N^e_i)^S$ *from* F^S_i *and* $n \in D$ **then**
12 $n_{loc} = D(n)$ -- is the same concept node in local schema retrieved from D ;
13 Replace n_{loc} in S'_L with variable var ;
14 $BGP^Q_L = Convert(S')$ -- replace summary nodes in S' with generated variables ;
15 $Q_L = REPLACE(Q, BGP^Q, BGP^Q_L)$ -- replace the basic graph pattern in query Q;
16 $RD = RD \cup \{\gamma(vars_\pi(Q))\} \setminus \{f(vars_\pi(Q))\}$ where γ is a rewrite of the aggregate function f;
17 **return** Q_L ;

Algorithm 1: Query rewriting using a global-to-local mapping

```
  ?obs qb:dataSet ex:Precipitation .
  ?obs ex:precipitationLevel ?prec .
  ?obs ex:station ?station .
  ?station qb4o:inLevel ex:Station .
  ?station skos:broader ?loc .
  ?loc qb4o:inLevel ex:AdmDivLevel2 .
  ?obs sen:samplingTime ?date .
  FILTER(?date='2014-02-07'^^xsd:date)
} GROUP BY ?loc
```

Listing 4.3: Global query

```
SELECT ?loc (SUM(?node6) as ?totalPrec)
WHERE{ ?node1 rdf:type ae:Observation .
  ?node1 ae:valueOfObservedData ?prec .
  ?node1 ssn:observedBy/ssn:locatedIn ?node9 .
  ?node9 owl:sameAs ?node12 .
  SERVICE <http://lod2.openlinksw.com/sparql> {
```

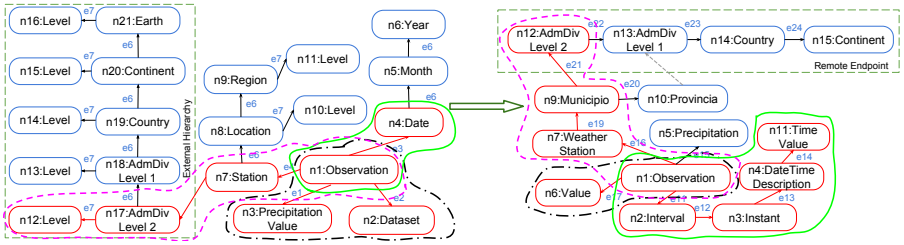


Fig. 4.4: Query rewriting

```
    ?node12 gn:name ?loc . }
  ?node1 ssn:observationSamplingTime ?node6 .
  ?node6 tm:inInterval/tm:hasBeginning ?node3 .
  ?node3 tm:inDateTime/tm:inXSDDateTime ?date .
  FILTER(?date='2014-02-07'^^xsd:date)
} GROUP BY ?loc
```

Listing 4.4: Local query

Let us consider an example. Listing 4.3 shows a global query that calculates the total precipitation at each location on a specific day and groups the results by the location. This query is executed against the global schema and the resulting subgraph is shown in red in Figure 4.4. Based on the resulting subgraph, we find the corresponding schema fragments (represented by solid green line, dashed magenta and black lines) and its counterparts for the local schema (represented by the same color lines). We assemble the local schema fragments and build a subgraph corresponding to a query over the local dataset (in red). Then, based on the resulting subgraph and the same concept mapping, we build a query over the local dataset. The resulting local query is shown in Listing 4.4. However, in the federated environment it is important to optimize the query before the execution.

5.2 Global and Local Optimization

There are two basic options to evaluate queries with externally defined hierarchies: (a) to execute *SERVICE* queries for every observation while evaluating a query in a nested loop join (NLJ) fashion or (b) to download all triples matching the specified graph pattern of the *SERVICE* query for evaluating the query locally. In the approach (a), the NLJ causes many remote requests (one request per observation, although the batches of bindings can be combined for optimization) while in the approach (b) a large amount of potentially irrelevant data needs to be shipped from the remote endpoint.

Thus, the system intends to optimize queries based on globally available information first. We use rule-based optimization on the global level and cost-based optimization on the local level. On the global level, we opted

for rule-based optimization due to the complexity of calculating estimations in federated systems [100], although LITE can be extended with cost-based optimization in the presence of necessary statistics.

On the global level, we try to (1) use internal hierarchies for rolling up the hierarchy levels if possible, (2) place restrictions on external hierarchies to retrieve only necessary results, or (3) roll-up data for external hierarchies to the maximal hierarchy levels on local endpoints instead of rolling-up the data in a mediator. These rules are designed to reduce the amount of transferred data and increase the performance of the system.

For rule (1), we check if the system can use internal hierarchies to roll-up to a level that corresponds to some level of the external hierarchy. For example, Rainfall Districts in Australia do not correspond to administrative districts recorded in GeoNames but Rainfall States (the next hierarchy level) are the same as administrative states. Thus, if the query aggregates on the state level or above, the system can use local data to roll-up to the state level, and external hierarchies to roll-up further. For Australia, 14 times less data is received from GeoNames when this rule is applied.

Input: schema S'_L that corresponds to Q_L, mapping schema M
Output: Optimized schema S_{Opt}

1 $F'_E = [F_{E1}, \ldots, F_{Ek}]$; $F'_E \in M$ -- is a set of schema fragments in the local schema S'_L that are external to the local source ;
2 $D' = \cup_{i=1,\ldots,m}(n_i^E, n_i^L)$ -- is a dictionary of same concept nodes for local and external fragments F_L, F_E from mapping M ;
3 From $F_L \in S_L$ find a subset $F'_L = [F_{L1}, \ldots, F_{Lm}]$ such that F'_L is a roll-up path and $\exists\ n \in F'_E$ and $n \in D'$ such that $\exists\ n_{loc} = D'(n)$ the same concept node and $n_{loc} \in F'_L(F_{Lm})$;
4 $F''_E = [F_{E1}, \ldots, F_{Ek}]$; $F''_E \subseteq F'_E$; $n \in F_{Ek}$; -- is a part of external hierarchy roll-up path (as a connected graph) with the same concept node n;
5 S_{Opt} = Replace F''_E in S'_L with F'_L ;
6 $F_E = [F_{E1}, \ldots, F_{Ex}]$; $F_E \in M$ -- is a set of all schema fragments in the local schema S_L that are external to the local source ;
7 R -- is a set of restricted values in F_E that are external to the local source ;
8 $F'''_E = [F_{Ek+1}, \ldots, F_{Ex}]$ -- is a set of schema fragments that form a roll-up path such that $F''_E \cup F'''_E$ is also connected and $R \in F'''_E$;
9 $S_{Opt} = S_{Opt} \cup F'''_E$;
10 **return** S_{Opt} ;

Algorithm 2: Optimizing query

For rule (2), we place a restriction on the values for hierarchy levels so that the *SERVICE* query does not retrieve irrelevant data. For example, if the endpoint contains precipitation data related to Australia, a remote query to GeoNames will retrieve Australia-related data only (using *FILTER* clause),

significantly reducing the amount of transferred data (559 result bindings instead of 40128 for districts in Australia).

For rule (3), we intend to reduce the amount of data transferred from the local endpoint to a mediator. For example, if the local datasets provide information on the City level and the user query asks to roll-up to the Continent level, we can either roll-up data to the Continent level on local endpoints and merge the results on the mediator, or aggregate on each endpoint by the City level and perform the roll-up on the mediator. In general, the amount of data transferred from the local sources to the mediator is less in the first approach (1 result for a continent versus 99 aggregations for Rainfall Districts for Australia) while the aggregation of data can happen in parallel in a number of endpoints. The optimization algorithm is given in Algorithm 2.

After applying rule-based optimization on the global level, we use CoDA [69] for cost-based optimization on the local level. Based on estimating constants and result sizes for triple patterns, joins, grouping and aggregation, CoDA uses a cost model to select among different query execution strategies like Mediator Join, SemiJoin, or Partial Aggregation for queries with *SERVICE* construct. The local endpoints then execute optimized queries. More information on CoDA is available in [69].

However, before executing the query over a local dataset we should account for implicit hierarchies that may appear due to the RDFS entailment (Sec. 5.3).

5.3 RDF Entailment

A valuable feature of RDF is RDF Schema (RDFS). RDFS is used to enhance the meaning of an RDF dataset – it defines the semantic constraints between the classes and properties used for resource descriptions. These constraints are stated using the RDFS standard properties *subClassOf*, *subPropertyOf*, *domain*, and *range*. As a result, RDFS entails implicit triples that may not be explicitly present in RDF dataset. Accounting for the implicit triples is necessary for returning a complete answer.

In our framework, we are interested in accounting for *hierarchical* information present in the data. We handle dimensional hierarchies by rewriting the queries based on the specified mappings. However, local data sources may contain hierarchical information encoded using predefined RDFS properties, such as *rdfs:subClassOf* or *rdfs:subPropertyOf*. These hierarchies should also be taken into consideration.

For example, a person in DBpedia belongs to several classes including *dbo:Person*. Some of these classes denote a profession of the person like *dbo:Journalist* or *dbo:Actor*. The classes denoting the profession have a subclass relation (*rdfs:subClassOf*) with *dbo:Person*. Thus, we can build a new hierarchy among the persons with two hierarchy levels – Profession and All

– using solely RDFS entailment.

There are two main approaches for processing queries when considering RDF entailment. In the dataset saturation approach, all implicit triples are materialized and added to the dataset. While requiring more space and complex maintenance, this method benefits from applying plain query evaluation techniques to compute the answer. Query reformulation, on the other hand, leaves the dataset unchanged but reformulates a query as a union of queries and increases the overhead during query evaluation. Since the data sources are independent in a federated setup, we use the query reformulation approach in our framework.

Input: RDFS schema S, Query Q
Output: Rewritten query Q'

1 $T = BGP(Q)$ -- a basic graph pattern of query Q ; $T' = T$;
2 **foreach** $t(s, p_1, o) \in T$ **do**
3 **if** $(p_1, rdfs{:}subPropertyOf, p_2) \in S$ **then**
4 $T' = T' \cup UNION\{newt(s, p_2, o)\}$ -- new triple pattern ;
5 **foreach** $t_i(s_1, rdf{:}type, X) \in T$ *and* $t_j(s_2, rdf{:}type, Y) \in T$ **do**
6 **if** $(X, rdfs{:}subClassOf, Y) \in S$ **then**
7 $T' = T' \cup UNION\{newt(s_1, newVar(p), s_2)\}$ -- new triple pattern ;
8 Q' = $REPLACE(Q, T, T')$ -- replace T with T' in query Q;
9 **return** Q' ;

Algorithm 3: Query rewriting considering RDFS entailment

Algorithm 3 is used for rewriting a query considering RDFS entailment. For all rules specifying hierarchical relations between predicates (*rdfs:subPropertyOf*), the algorithm adds an *UNION* triple pattern for triples with the implicit predicates specified in rules (lines 2–4). Then, for all rules specifying the hierarchical relations between classes (*rdfs:subClassOf*), the algorithm adds an *UNION* triple pattern with two hierarchically connected nodes and a newly generated variable replacing the predicate (lines 5–7). After rewriting the query, it can be passed for further execution to the endpoint containing the local dataset.

6 Experimental Evaluation

To evaluate the performance gain of LITE for federated analytical queries over existing systems, we implemented LITE using the .NET Framework 4.0 and the dotNetRDF (`http://dotnetrdf.org/`) API. We report total response time, i.e., including query rewriting (typically very small, on average 3%) and query execution. The timeout was set to 30 minutes for each query. All

queries were executed 5 times following a single warm-up run. The average runtime is reported for all queries.

For comparison, we need a system that fully support *both* aggregate and federated SPARQL queries. Thus we initially considered with three open source systems that fully support the SPARQL1.1 specification (Virtuoso, Sesame, and Jena). Here, Jena and Virtuoso use the same strategy to evaluate federated queries, while Sesame tries to download all matched triple patterns from the remote endpoint and times out even for small datasets, and was thus discarded. Jena could not efficiently load the required data volumes, so we ultimately chose Virtuoso as the comparison target.

6.1 Datasets, Setup, and Queries

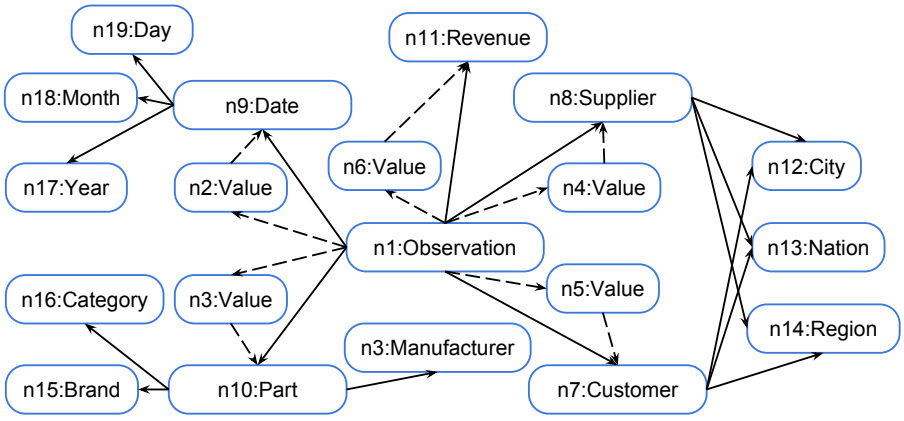


Fig. 4.5: SSB Dataset

Unfortunately, none of the standard SPARQL benchmarks are applicable to our scenario. Existing benchmarks either do not define analytical SPARQL queries or do not require linked data from other endpoints to answer the queries. Therefore, we decided to extend the Star Schema Benchmark (SSB) [66], originally designed for aggregate queries in relational database systems. This benchmark is well-known in the database community and was chosen for its well-defined testbed and its simple design.

The data in the SSB benchmark represent sales in a retail company; each transaction is defined as an observation described by 4 dimensions (Parts, Customers, Suppliers, and Dates). We translated the data into the RDF representation as illustrated in Figure 4.5. An observation is connected to dimensions (objects) via certain predicates. The Suppliers and Customers dimension contain information about cities, countries and world regions for each supplier/customer. We linked each city/country present in the dataset

to their counterparts from the GeoNames dataset using *owl:sameAs* predicate, thus showing how external hierarchies can be added. To establish a federated setup, we divided the data among 5 SPARQL endpoints, each storing observations for one of the world regions defined in SSB: Africa, America, Asia, Europe and Middle East. The schema in each SPARQL endpoint was made slightly different from the other schemas, by generating an intermediary graph node between the observation and one of the dimensions or the value of revenue (dashed lines in Figure 4.5) for every SPARQL endpoint. For example, the data schema in the Africa endpoint was different in the Parts dimension and so on. Each endpoint was hosted on a separate physical machine running 64-bit Debian "jessie" 8.7 with Intel(R) Core 2 Duo E8400 CPU, 4GB RAM, 160GB HDD and Virtuoso v06.01.3127 as triple store. The data related to the GeoNames dataset (GeoNames dump) were held on a separate machine running 64-bit Ubuntu 14.04 LTS with Intel(R) Core(TM) i7-950 CPU, 24GB RAM, 600GB HDD. All machines were located on the same LAN (the performance gains would be even higher on a WAN). We used scaling factors 1 to 3 to obtain datasets of different sizes (120 to 360M triples). SSB defines 13 classic data warehouse queries that are typical in business intelligence scenarios. We converted all 13 queries into SPARQL. All queries, schemas, and datasets are available at `http://extbi.cs.aau.dk/fedsystem`.

6.2 Query Evaluation

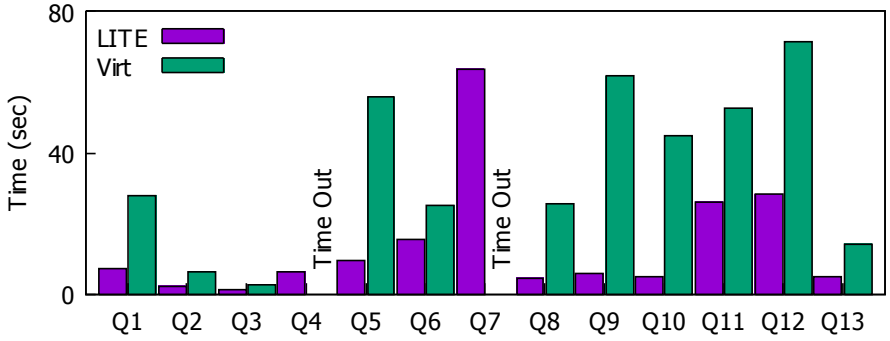
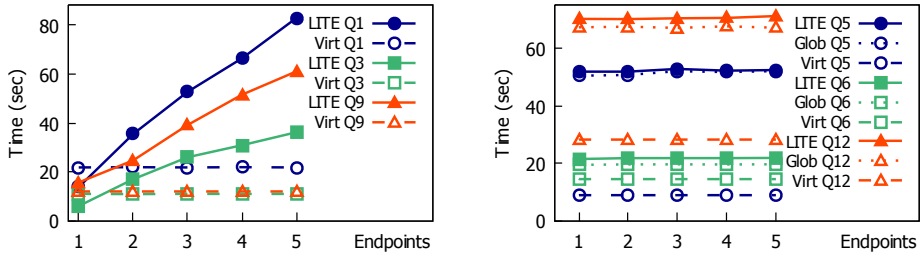


Fig. 4.6: Performance of queries over scale factor 1

Figure 4.6 shows the results of executing the SSB queries in LITE and Virtuoso over scale factor 1. LITE performs on average 4 times faster than plain Virtuoso (where some queries even time out). The only exception is Q12 which has a very large result size with 7600 mappings, yielding less effect of the optimization.

We also evaluated how the number of endpoints in the federation in-



(a) Queries 1, 3,and 9 **(b)** Queries 5, 6, and 12

Fig. 4.7: Comparing executions of queries

fluence the performance of the queries (Figure 4.7 – selected queries with similar result patterns are grouped into separate graphs for better visualization). The solid lines show the execution of queries in Virtuoso, dashed lines of the same color show the execution of the queries using LITE, while dotted lines show the execution after applying only global optimization. The execution time for plain Virtuoso in Figure 4.7a grow linearly with the number of endpoints, as it needs query more endpoints to retrieve the result, while LITE times are constant due to parallel execution. For one endpoint, Virtuoso slightly outperforms LITE for Q1 and Q3 due to the small parsing/optimization overhead. Also, Q1 and Q3 do not retrieve dimensional data from the external endpoint, so the query optimization does not have any effect on these queries. On the other hand, the execution time for queries in Figure 4.7b is not affected by the number of endpoints. . The *FILTER* expression of these queries apply restrictions on the continent value therefore data from one endpoint is enough to answer these queries. Thus, the requests to other endpoints return no results and the overall execution time for the queries do not change. The times for LITE queries stay the same regardless of the number of endpoints due to the parallel execution of queries over several endpoints.

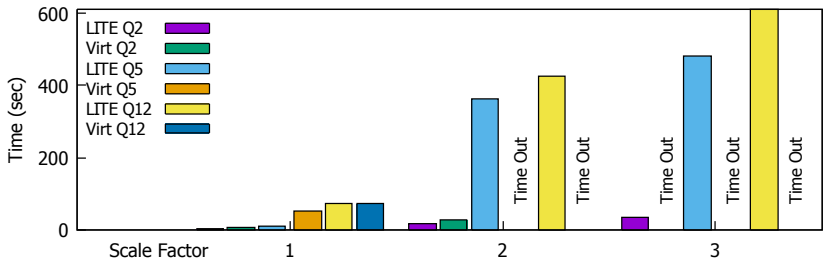


Fig. 4.8: Comparison of queries over scale factors

In the next line of experiments, we measured the runtime for the benchmark queries over different scale factors to evaluate the influence of the volume of data to the execution of the queries (Figure 4.8). For plain Virtuoso queries, the system performance quickly degrades and starting from scale factor 2, the queries time out due to the increased amount of data that needs to be processed, and the increased number of external requests to the GeoNames dataset. The performance of queries with higher selectivity that do not retrieve dimensional data from the GeoNames dataset degraded slower (Q2 was successfully executed on scale factor 2 data). In comparison, LITE queries were executed successfully for all tested data sizes and were much faster overall.

In summary, the experimental results show that LITE can significantly optimize analytical queries in a federated setup. The advantage of LITE is even more evident for queries that retrieve hierarchical data from remote endpoints. The experiments also show that evaluating queries using LITE is much faster (up to 7x times faster) than on current systems (Virtuoso) and scales well when the data volumes and number of endpoints grow.

7 Conclusion

In this chapter, we have addressed the problem of efficiently evaluating analytical SPARQL queries over a federation of SPARQL endpoints. To solve this problem, we have proposed LITE (O**L**AP-style Analyt**I**cs in a Federa**T**ion of SPARQL **E**ndpoints) and presented its main components. In particular, LITE enables RDF-based schema mappings and comes with algorithms for query rewriting and optimization in consideration of RDF specifics. A comprehensive experimental evaluation showed the efficiency and scalability of the proposed approach. In our future work, we plan to develop a cost model for optimizing queries on a global level based on some statistical data, build the approach that accounts for data overlap in sources, and investigate the potential of considering materialized RDF views to increase performance even further.

Chapter 5

Optimizing Aggregate SPARQL Queries Using Materialized RDF Views

Dilshod Ibragimov, Torben Bach Pedersen, Katja Hose and Esteban Zimányi

The paper has been published in the
Proceedings of the 15th International Semantic Web Conference, Kobe, Japan, pp. 341–359, 2016. DOI: 10.1007/978-3-319-46523-4_21
The final publication is available at Springer via `http://dx.doi.org/10.1007/978-3-319-46523-4_21`

Ibragimov D., Hose K., Pedersen T.B., Zimányi E. Optimizing Aggregate SPARQL Queries Using Materialized RDF Views. In: Groth P. et al. (eds) The Semantic Web – ISWC 2016. ISWC 2016. Lecture Notes in Computer Science, vol 9981. Springer, Cham, 2016
The layout of the paper has been revised.

Abstract

During recent years, more and more data has been published as native RDF datasets. In this setup, both the size of the datasets and the need to process aggregate queries represent challenges for standard SPARQL query processing techniques. To overcome these limitations, materialized views can be created and used as a source of precomputed partial results during query processing. However, materialized view techniques as proposed for relational databases do not support RDF specifics, such as incompleteness and the need to support implicit (derived) information. To overcome these challenges, this chapter proposes MARVEL (MAterialized Rdf Views with Entailment and incompLetness). The approach consists of a view selection algorithm based on an associated RDF-specific cost model, a view definition syntax, and an algorithm for rewriting SPARQL queries using materialized RDF views. The experimental evaluation shows that MARVEL can improve query response time by more than an order of magnitude while effectively handling RDF specifics.

1 Introduction

The growing popularity of the Semantic Web encourages data providers to publish RDF data as Linked Open Data, freely accessible, and queryable via SPARQL endpoints [7]. Some of these datasets consist of billions of triples. In a business use case, the data provided by these sources can be applied in the context of On-Line Analytical Processing (OLAP) on RDF data [68] or provide valuable insight when combined with internal (production) data and help facilitate well-informed decisions by non-expert users [5].

In this context, new requirements and challenges for RDF analytics emerge. Traditionally, OLAP on RDF data was done by extracting multidimensional data from the Semantic Web and inserting it into relational data warehouses [28]. This approach, however, is not applicable to autonomous and highly volatile data on the Web, since changes in the sources may lead to changes in the structure of the data warehouse (new tables or columns might have to be created) and will impact the entire Extract-Transform-Load process that needs to reflect the changes. In comparison to relational systems, native RDF systems are better at handling the graph-structured RDF model and other RDF specifics. For example, RDF systems support triples with *blank* nodes (triples with unknown components) whereas relational systems require all attributes to either have some value or *null*. Additionally, RDF systems support entailment, i.e., new information can be derived from the data using RDF semantics while standard relational databases are limited to explicit data.

Processing analytical queries in the context of Linked Data and federations of SPARQL endpoints has been studied in [69,70]. However, performing

aggregate queries on large graphs in SPARQL endpoints is costly, especially if RDF specifics need to be taken into account. Thus, triple stores need to employ special techniques to speed up aggregate query execution. One of these techniques is to use materialized views – named queries whose results are physically stored in the system. These aggregated query results can then be used for answering subsequent analytical queries. Materialized views are typically much smaller in size than the original data and can be processed faster.

In this chapter, we consider the problem of using materialized views in the form of RDF graphs to speed up analytical SPARQL queries. Our approach (MARVEL) focuses on the issues of selecting RDF views for materialization and rewriting SPARQL aggregate queries using these views. In particular, the contributions of this chapter are:

- A cost model and an algorithm for selecting an appropriate set of views to materialize in consideration of RDF specifics
- A SPARQL syntax for defining aggregate views
- An algorithm for rewriting SPARQL queries using materialized RDF views

Our experimental evaluation shows that our techniques lead to gains in performance of up to an order of magnitude.

The remainder of this chapter is structured as follows. Section 2 discusses related work. Section 3 introduces the used RDF and SPARQL notation and describes the representation of multidimensional data in RDF. Section 4 specifies the cost model for view selection, and Section 5 describes query rewriting. We then evaluate MARVEL in Section 6, and Section 7 concludes the chapter with an outlook to future work.

2 Related Work

Answering queries using views is a complex problem that has been extensively studied in the context of relational databases [71]. However, as discussed in [71, 72], aggregate queries add additional complexity to the problem.

In relational systems, the literature proposes semantic approaches for rewriting queries [72] as well as syntactic transformations [73]. However, SPARQL query rewriting is more complex. The results for views defined as *SELECT* queries represent solutions in tabular form, so that the solutions need to be converted afterwards into triples for further storage, thus making a view definition in SPARQL more complex and precluding view expansion (replacing the view by its definition).

Another problem in this context is to decide which views to materialize in order to minimize the average response time for a query. [74] addresses this problem in relational systems by proposing a cost model leading to a trade-off between space consumption and query response time for an arbitrary set of queries. [75] provides a method to generate views for a given set of select-project-join queries in a data warehouse by detecting and exploiting common sub-expressions in a set of queries. [76] further optimizes the view selection by automatically selecting an appropriate set of views based on the query workload and view materialization costs. However, these approaches have been developed in the context of relational systems and, therefore, do not take into account RDF specifics such as entailment, the different data organization (triples vs. tuples), the *graph*-like structure of the stored data, etc.

The literature proposes some approaches for answering SPARQL queries using views. [77] proposes a system that analyzes whether query execution can be sped up by using precomputed partial results for conjunctive queries. The system also reduces the number of joins between tables of a back-end relational database system. While [77] examines core system improvements, [78] considers SPARQL query rewriting algorithms over a number of virtual SPARQL views. The algorithm proposed in [78] also removes redundant triple patterns coming from the same view and eliminates rewritings with empty results. Unlike [78], [79] examines materialized views. Based on a cost model and a set of user defined queries, [79] proposes an algorithm to identify a set of candidate views for materialization that also account for implicit triples. However, these approaches [77–79] focus on conjunctive queries only. The complexity of loosing the multiplicity on grouping attributes (by grouping on attribute X, we loose the multiplicity of X in data) and aggregating other attributes is not addressed by these solutions.

The performance gain of RDF aggregate views has been empirically evaluated in [26], where views are constructed manually and fully match the predefined set of queries. Hence, the paper empirically evaluates the performance gain of RDF views but does not propose any algorithm for query rewriting and view selection.

Algorithms that use the materialized result of an RDF analytical query to compute the answer to a subsequent query are proposed in [80]. The answer is computed based on the intermediate results of the original analytical query. However, the approach does not propose any algorithm for view selection. It is applicable for the subsequent queries and not to an arbitrary set of queries.

Although several approaches consider answering queries over RDF views [77–79], none of them considers analytical queries and aggregation. In this chapter, we address this problem in consideration of RDF specifics such as entailment and data organization in the form of triples, and taking into account the graph structure of the stored data. In particular, this chapter pro-

poses techniques for cost-based selection of materialized views for aggregate queries, query rewriting techniques, and a syntax for defining such views.

3 RDF Graphs and Aggregate Queries

The notation we use in this chapter follows [81] and is based on the three disjoint sets of blank nodes B, literals L, and URIs/IRIs U. For simplicitly we abbreviate the union of them as BLU.

An RDF [2] triple $(s, p, o) \in BU \times U \times BLU$ connects a subject s through predicate p to object o. An RDF database (G) consists of a finite set of triples and is often represented as a graph. Queries are based on graph patterns that are matched against G. A Basic Graph Pattern (BGP) P consists of a set of triple patterns of the form $(UV) \times (UV) \times (LUV)$, where V ($V \cap BLU = \emptyset$) is a set of query variables. We distinguish variables by a leading question mark symbol, e.g., ?x or ?y, and literals by a leading pound symbol, e.g., #o. We denote a set of variables occurring in a graph pattern P as $vars(P)$.

A solution mapping is a mapping μ from a set of variables V to a set of RDF terms BLU, i.e., $\mu : V \rightarrow BLU$. A solution sequence is a list of solutions. A BGP can be extended by additional conditions in a *FILTER* expression to restrict solutions to only those for which the filter expression evaluates to true.

Similar to relational query languages, an *aggregate* function uses an aggregate expression to compute a scalar value over groups of solutions. A *group* function groups a solution sequence into multiple solutions based on a given list of attribute(s).

Most common SPARQL 1.1 [47] aggregate queries conform to the form *SELECT* RD *WHERE* GP *GROUP BY* GRP, where RD is the result description based on a subset of variables in the graph pattern GP. GP contains BGP and optional functions, such as *assignment* functions (e.g., *BIND* and assignment expressions) and constraints (e.g., *FILTER*). GRP defines a set of grouping variables ($vars_{GRP}(GP) \subset vars(GP)$), whereas RD contains selection description variables ($vars_{SEL}(GP) \subseteq vars_{GRP}(GP)$) as well as aggregation variables ($vars_{AGG}(GP) \subset vars(GP)$) with corresponding aggregate functions. In this chapter, we consider the standard aggregate functions *COUNT*, *SUM*, *AVG*, *MIN*, and *MAX*).

A BGP can be represented as a directed labeled multi-graph whose nodes N correspond to subjects and objects in the triple patterns. The set of edges E contains one edge for each triple pattern in the BGP with the predicate as a label. In data analytics, graph patterns of SPARQL queries have a special, *rooted* pattern [68]. A BGP is rooted in node $n \in N$ iff any node $x \in N$ is reachable from n following directed edges in the graph. The graph patterns of the queries are rooted due to the typical graph structure of the data that

are processed for data analysis. It is based on the multidimensional model, which may represent data in an *n*-dimensional space, called a *data cube* or a *hypercube*. A data cube is defined by *dimensions* (perspectives used to analyze the data) and *observations* (facts). Dimensions are structured in *hierarchies* to allow analyses at different aggregation levels. Dimension level instances are called *members*. Data cube cells (observations) that are analyzed along the dimensions have associated values called *measures*, which can be aggregated.

A typical example of multidimensional RDF data is the statistical data published using the W3C RDF Data Cube Vocabulary (QB) [30]. QB datasets consist of a collection of observations characterized by a set of dimensions defining what the observation applies to, along with measure components representing the phenomenon being observed, and metadata describing the measures. A set of values for all the dimension components can identify a single observation, and is usually attached to the observation through properties. This observation structure is called a normalized structure.

An example of data with hierarchical dimensions is the utilities consumption data from electricity and gas meters for Scottish Government buildings[1]. The data is available as energy usage over a daily period. The unit used for the data is kilowatt-hour (gas consumption is converted to kilowatt-hours using standard conversion factors). Figure 5.1 sketches the data with two hierarchical dimensions where Building, Locality and Region are hierarchy levels in Geography dimension, Repodt Date, Month and Year are hierarchy levels in Date dimension, and the observation contains a utility consumption measure (connected to Data node in Figure 5.1).

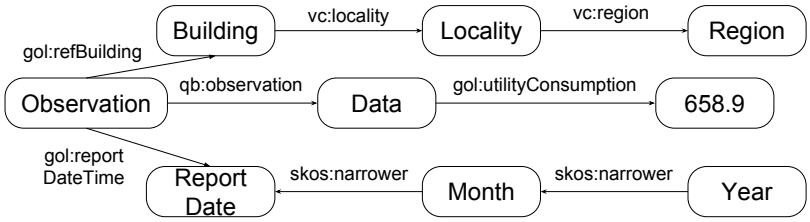


Fig. 5.1: Representing Observations in RDF

A dataset with such observations is stored in a SPARQL endpoint[2] to enable analytical querying involving aggregations and groupings on different hierarchy levels. For example, the following query computes the daily consumption of electricity in each city in September 2015:

```
SELECT ?date ?place (SUM(?val) as ?value)
WHERE {
```

[1]http://cofog01.data.scotland.gov.uk/id/dataset/golspie/utilities
[2]http://cofog01.data.scotland.gov.uk/sparql

```
  ?obs gol:refBuilding ?bld ;
    gol:reportDateTime ?date ; qb:observation ?data .
  ?data  gol:utilityConsumption ?val .
  ?bld org:siteAddress/vc:adr/vc:locality ?place .
  ?month skos:narrower ?date . ?month gol:value ?mVal .
  FILTER (?mVal = 'September 2015')
} GROUP BY ?date ?place
```

Listing 5.1: Example Query with Grouping and Aggregation

Another example of multidimensional data (not published as QB) is the statistics on radioactivity observations published after a nuclear accident in Japan in March 2011 that was triggered by an earthquake and subsequent tsunami. The daily announcements of radioactivity statistics observed hourly at 47 prefectures from March 16, 2011 to March 15, 2012 were converted to RDF data and made publicly available via a SPARQL endpoint (`http://www.kanzaki.com/works/2011/stat/ra/`). Figure 5.2 visualizes this dataset with its two hierarchical dimensions.

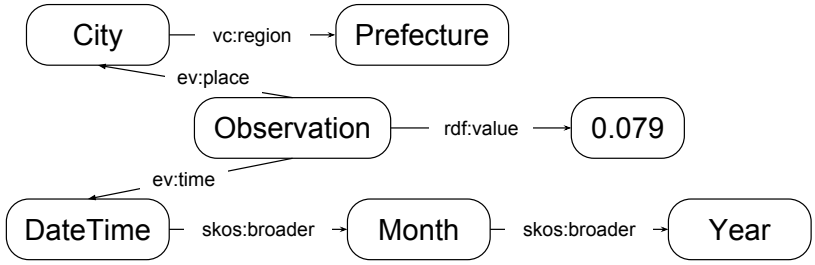


Fig. 5.2: Radioactivity Statistics in RDF

An example of an analytical query for this dataset is to compute the average radioactivity separately for each prefecture in Japan to find out which prefectures were more affected than others.

```
SELECT ?regionID (AVG(?radioValue) AS ?average)
WHERE {
  ?s ev:place ?placeID; ev:time ?time;
  rdf:value ?radioValue . ?placeID vc:region ?regionID .
} GROUP BY ?regionID
```

Listing 5.2: Example Query with Grouping and Aggregation

Traditional OLAP operations associated with multidimensional data cubes are roll-up (increasing the level of aggregation), drill-down (decreasing the level of aggregation or increasing detail), and slicing and dicing (selection and projection). These operations transform a cube specified by a SPARQL query into another.

Slice The slice operation removes a dimension in a cube by fixing a single value in the dimensions hierarchy level – a cube of $n - 1$ dimensions is

obtained from a cube of n dimensions. The other dimensions remain unchanged. Given the query Q, the slice in SPARQL can be achieved by constraining the value of a certain variable to the specified value, e.g. by using *FILTER* operator.

Dice The dice operation keeps in a cube only those cells that satisfy a specific Boolean condition. This condition is imposed over dimension levels, attributes, and measures. Intuitively, the dice operation restricts several aggregation dimension to specific set of values. It is analogous to the relational algebra selection, where the argument is a cube instead of a relation. Like in slice, the dice in SPARQL can be achieved by projection in combination with constraining the values of certain variables, e.g by using *FILTER* or *VALUES* operators. However, as the *VALUES* operator is not yet widely supported [19], *FILTER* must often be used.

Roll-up The roll-up operation aggregates measures at a coarser granularity along a given dimension. Given the hierarchical structure of the data in the graph, navigating up the hierarchy corresponds to adding classifier triples. Thus, for a given query Q, the roll-up in SPARQL can be achieved by adding to the graph pattern of the query the connected triple patterns that reflect the hierarchy structure in the dataset.

Drill-down The the drill-down operation – the inverse of roll–up – disaggregates previously summarized data to a child level in order to obtain measures at a finer granularity. In the hierarchical data, navigating down the hierarchy corresponds to removing classifier triples and for a given query Q, the drill-down in SPARQL can be achieved by removing some of the connected triple patterns that reflect the hierarchy structure in the dataset from the graph pattern of the query.

However, executing OLAP queries requires considerable amounts of processing resources at the endpoint, which becomes a bottleneck with with increasing amounts of data. To enable scalable processing, this chapter proposes RDF-specific techniques for materialized views. Hence, we define a *materialized view* as a named graph described by a query whose results are physically stored in a graph database in the form of triples. Given a query, the DBMS query optimizer checks whether the query can be answered based on one of the available materialized views. As the materialized views are typically much smaller than the original data, this can yield a significant performance boost. Precalculating all possible aggregations over the different dimension levels is infeasible, as it typically requires much more space than the original data [24]. Hence, it is very important to find an appropriate selection of views to materialize in order to minimize the total query response time and the cost of maintaining the selected views.

4 View Materialization in MARVEL

A high number of triples needs to be processed for evaluating OLAP queries on a dataset. This imposes high execution costs, especially when the amount of data increases. To enable scalable processing, we propose RDF-specific techniques to select a set of materialized views that can be used to evaluate queries more efficiently. We define a materialized view as a named graph described by a query whose results are physically stored in a triple store. Given a query, the system checks whether the query can be answered based on the available materialized views. As materialized views are typically smaller than the original/raw data, this can yield a significant performance boost. Precalculating all possible aggregations over all dimension levels is usually infeasible as it requires much more space than the raw data [24]. Thus, it is important to find an appropriate set of materialized views to minimize the total query response time.

4.1 Creating Materialized RDF Views

SQL has a standard mechanism to define a view, namely *CREATE VIEW*, which specifies the name of the view and the SQL query defining it. Since such a mechanism do not yet exist for SPARQL, we use *CONSTRUCT* queries (i.e., queries that output RDF graphs) to define views. Indeed, in SPARQL, the result of a *SELECT* query is a set of values in a tabular form, not a set of triples, which is what we need to store the result as RDF data. *CONSTRUCT* queries allows us to create RDF triples from the results of a *SELECT* subquery. For this, we need to specify the subject of the triples and the predicates connecting the subject to the query results. Since the combination of values in the *GROUP BY* clause of a *SELECT* subquery is unique, we can use these values to construct the subject of the triples in the *view defining CONSTRUCT* query.

Listing 5.3 gives an example of such a query, where the query in the *SELECT* clause aggregates utility consumptions by City and Date. We use the *IRI* function to create a resource identifier *id* based on the unique combination of City and Date. Note also the use of *STRAFTER* function to strip the protocol identifier *http://* from the *IRI* of the resource. The *CONSTRUCT* clause then creates triples by connecting the *id* to the resulting aggregate and grouping values.

```
CONSTRUCT {
  ?id gol:reportDate ?date ; gol:reportLocality ?vCity ;
    gol:utilityConsumption ?value .
} WHERE {
  SELECT ?id ?date ?vCity (SUM(?cons) as ?value)
  WHERE { ?fact gol:refBuilding ?bld ;
    gol:reportDateTime ?date ; qb:observation ?data .
   ?data  gol:utilityConsumption ?cons .
```

```
    ?bld org:siteAddress/vc:adr/vc:locality ?vCity .
    BIND(IRI('http://ex.org/id#', CONCAT(STRAFTER(STR(?dt),
      'http://'), STRAFTER(STR(?vCity), 'http://'))) AS ?id).
  } GROUP BY ?id ?date ?vCity
}
```

Listing 5.3: Query to Construct Materialized View

Examples of triples created by the query in Listing 5.3 are given in Listing 5.4. Values for City and Date and aggregated values are connected to the generated identifier. The set of all such triples forms the complete materialized view.

```
gol:City_11Date_01052013 gol:utilityConsumption 1019794;
  gol:reportLocality <http://data.gov.uk/.../AB0112>;
  gol:reportDate <http://data.gov.uk/.../20130501> .
gol:City_8Date_17092014 gol:utilityConsumption 1460796;
  gol:reportLocality <http://data.gov.uk/.../AB0099>;
  gol:reportDate <http://data.gov.uk/.../20140917> .
  .....
```

Listing 5.4: Materialized View Representation

4.2 Storing Materialized RDF Views

We now consider how to store the materialized RDF views. As stated in [47], an RDF dataset represents a collection of graphs. The RDF dataset comprises one default (nameless) graph and zero or more IRI-named graphs.

One option is to store all materialized views as well as the original base data in the default graph and specify the type of the data (base data or view data) in one of the triples as a part of the BGP of the query, e.g., as (*?obs rdf:type ex:aggview1* .) for a view. However, as discussed below, a better option is to store each materialized RDF view in a separate named graph, which is what we propose in this chapter.

A SPARQL query can specify the dataset to be used for matching by using the *FROM* and/or the *FROM NAMED* clause. When specified, the dataset is added into the default graph and thus extend the search space. Using named graphs allows us to separate base and view data. Separating views by named graphs has a number of benefits. First, maintaining view data is easier. Indeed, SPARQL 1.1 allows us to delete graph data and insert new data. Thus, we can easily specify that certain updates affect this graph (view) only. Second, finding and aggregating the data stored in a named graph is faster. If data in named graphs are stored in quads (<*context*><*subject*><*predicate*><*object*>), the system can index the data by the *context* and provide faster access to the data. If the data are stored in separate tables, scanning a smaller table is much faster than scanning a bigger table. Thus, the system will execute queries faster. Finally, a separation of base and view data into different graphs facilitates the correctness of aggregation

results. By explicitly specifying the graphs, the SPARQL engine excludes irrelevant data from the default data space and executes aggregations over the appropriate data. This will ensure result correctness.

4.3 Data Cube Lattice

We first introduce the notion of a data cube lattice. In its simplest form, a node in the lattice represents an aggregation by a given combination of dimensions. A view, in the data cube lattice terminology, is defined by a query with the same grouping as in the corresponding node. For example, in case of 3 dimensions, Part (*P*), Customer (*C*) and Date (*D*), possible nodes (grouping combinations) are *PCD*, *PC*, *PD*, *CD*, *P*, *C*, *D* and *All* (all values are grouped into one group). Nodes are connected if a node *j* can be computed from *i* and the number of grouping attributes of *i* is the number of attributes of *j* plus one. In our example, the view that corresponds to node *PC* can be computed from the view that corresponds to node *PCD*. We denote this dependence relation as $PC \preceq PCD$ and we call view *PCD* the *ancestor* of view *PC* and view *PC* is the *predecessor* of view *PCD*.

In the presence of dimension hierarchies, grouping by a dimension breaks down to grouping by different hierarchy levels of that dimension, thus the number of all possible combinations increases. The total number of all different nodes in the lattice [82] can be calculated as: $\prod_{i=1}^{k}(h_i + 1)$, where h_i is the number of hierarchy levels in dimension i and $(h_i + 1)$ accounts for the top level *All*.

The concept of the data cube lattice is useful since user queries over multidimensional data can be evaluated using the nodes in the lattice. Given a query grouping (*GROUP BY*), the lattice node with the given grouping represents the best candidate view for answering the query. Since these views are smaller in size than the raw data, calculating the answer from the views will be cheaper than calculating it from raw data. Thus, to answer user queries we need to find an appropriate set of views so that the multidimensional queries posed against the data could be mapped to one of these materialized views.

The data cube lattice is used for selecting aggregated views in a relational framework [74]. However, this framework considers data that is hierarchical, complete, and complies to a predefined schema, and therefore cannot be directly applied to RDF graphs that lack these characteristics. Indeed, hierarchies in RDF may not be expressed explicitly. For example, a parent-child relation between two entities may be implied from the *rdfs:subClassOf* semantics between the *rdf:type* properties of these entities: for two triples (x *rdf:type* c_1) and (y *rdf:type* c_2), if the relationship between c_1 and c_2 is defined by RDFS semantics (c_1 *rdfs:subClassOf* c_2), then we need to account that x and y belong to the same hierarchy and x is a child of y.

Additionally, RDF data may be *incomplete*. For example, the canonical-

ized Ontology Infobox dataset from the DBpedia Download 3.8 contains birth place information for 266,205 persons (either as a country, a populated place like city or village, or both). However, out of 266,205 records, 16,351 records contain information only about the country of birth. Thus the information *available* in the source may not contain the information that holds in the world and, therefore, should *ideally* be present in the source. Accordingly, an incomplete data source is defined as a pair $\Omega = (G_a, G_i)$ of two graphs, where $G_a \subseteq G_i$. G_a corresponds the available graph and G_i is the ideal graph [83].

Definition *(Incomplete View).* A view is *incomplete* if its defining query over the available graph does not produce the same results as the defining query over the ideal graph: $[q_v]_{G_a} \neq [q_v]_{G_i}$.

Such incomplete views may not be used to answer queries involving the grouping over a higher hierarchy level than in the view. In the above example, the aggregation over the city of birth is incomplete and the city level view, due to incompleteness, cannot be used to roll-up to the country level even though the relationship City $\rightarrow$ Country between the levels holds.

In summary, for RDF data cubes we need to account for implicitly specified hierarchies, heterogeneity of data, and incompleteness of views. Therefore, we propose a novel aggregate view selection model that supports RDF-specific requirements unlike earlier models.

4.4 MARVEL Cost Model

MARVEL takes into account that RDF data are stored as triples and not as tuples. Thus, the cost of answering an aggregate SPARQL query in a generic RDF store is defined as the number of triples contained in the materialized view used to answer the query. This cost model is simple and works for a general case. More complex models that account for algorithms and auxiliary structures used in a particular triple store are certainly possible.

Usually, an observation in a view is described by its n dimensions. In addition, the observation is related to m measures, so that the number of triples for one observation is $(n + m)$. Thus, the size of a view w is equal to:

$$Size(w) = (n + m) * N \tag{5.1}$$

where N is the number of observations. This number is used to calculate the benefit of materializing the view. Note that the size of the view w serves as the cost of the view v if the view v is computed from the view w: *Cost(v) = Size(w)*. Note also that at the beginning the cost of every view is equal to the size of the base view (base data).

Let B_w be the benefit of view w. For every view v such that $v \preceq w$ the

benefit of view w relative to v is calculated as

$$\begin{aligned} &B_{w,v} = (Cost(v) - Size(w)) \text{ if } Cost(v) > Size(w). \\ &B_{w,v} = 0 \text{ otherwise.} \end{aligned} \tag{5.2}$$

That is, the difference between the current cost of the view v and the possible cost of the same view (if view w is materialized and used to compute the view v) contributes to the benefit of the view w. We then sum up the benefits for all appropriate views to receive the full benefit of the view w:

$$B_w = \sum B_{w,v_i} \text{ for all } i \text{ such that } v_i \preceq w. \tag{5.3}$$

Note that this value of benefit is absolute and does not take into account the size of the view. If the storage space is limited, the benefit of each view per *unit space* can be considered instead. In this case, the value of the benefit is calculated by dividing the absolute benefit of the view to its size:

$$B_w^{'} = \frac{B_w}{Size(w)} \tag{5.4}$$

In addition, we need a flexible way of defining a cube schema over RDF data since usually RDF data do not strictly conform to a predefined schema. Equally important is the possibility to specify that a hierarchy level should be computed from several ancestor levels (i.e., from a set of views) or should be complemented by observations that were not taken into account for ancestor levels. For example, for a birth place dimension we can specify that the roll-up to the Country level should be calculated from both the City and the Person levels since some people do not contain information about their city of birth but only about the country. Therefore, we use QB4OLAP schema [25] to describe the dataset and annotate it with information about the completeness of levels, about how to construct triples for the next hierarchy levels, and the types of hierarchy levels. We chose QB4OLAP over Analytical Schema (AnS) introduced in [68] since QB4OLAP allows representing OLAP cubes in RDF by adding the capability of representing dimension levels, level members, roll-up relations between levels and members, and associating aggregate functions with measures. AnS, on the other hand, underlines the structure of the needed data in a large dataset and allows to define perspectives (referred to as lenses) for analysis of RDF data but does not allow us to specify dimensions, complex hierarchies (for instance, as given above), or measures.

When a hierarchy level is computed from several ancestor levels, we say that the view corresponding to this level should be calculated from a *set* of views. We denote this dependence relation as $w \preceq \{v_i \dots v_n\}$. where w is the current view and $\{v_i \dots v_n\}$ are the ancestor views. In general, we can distinguish the following roll-up cases:

- **Single path roll-up**: a view w can be derived from either of the views $v_1 \dots v_n$, i.e. $\exists w, v_1 \dots v_n$ such that $w \preceq v_i$ and $v_i \not\preceq v_j$ for $i, j = \{1 \dots n\}$
- **Multiple path roll-up**: a view w can be derived from the union of views $v_1 \cup \dots \cup v_n$ while deriving w from any single v_i will be incomplete: $\exists w, v_1 \dots v_n$ such that $w \preceq \{v_i \dots v_n\}$, $w \not\preceq v_i$, and $v_i \not\preceq v_j$ for $i, j = \{1 \dots n\}$

Listing 5.5 provides an excerpt of the QB4OLAP schema showing the hierarchy steps that define the roll-up paths for the dp:CountryLvl level of the Person hierarchy. It shows that aggregations over the dp:CountryLvl level should be derived from both the aggregations by the dp:CityLvl level and the observation data since the aggregation over just one roll-up path will be insufficient (dp:isPartial "true"^^xsd:boolean).

```
dp:phs4 a qb4o:HierarchyStep ;
  qb4o:parentLevel dp:CountryLvl ;
  qb4o:childLevel dp:CityLvl ;
  rdf:predicate dbo:country ;
  dp:direction dp:forward ;
  dp:datatype dbo:City ;
  qb4o:cardinality qb4o:OneToMany .
dp:phs8 a qb4o:HierarchyStep ;
  qb4o:parentLevel dp:CountryLvl ;
  qb4o:childLevel qb:PersonLvl ;
  rdf:predicate dbo:birthPlace ;
  dp:direction dp:forward ;
  dp:datatype dbo:Country ;
  dp:isPartial "true"^^xsd:boolean ;
  qb4o:cardinality qb4o:OneToMany .
```

Listing 5.5: Defining Hierarchy Steps

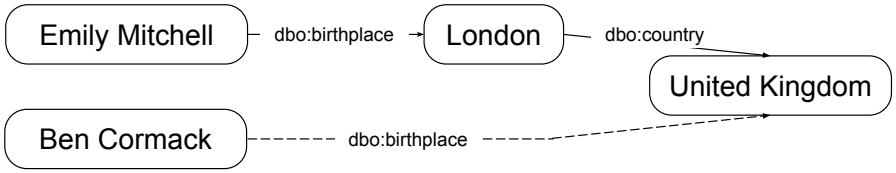


Fig. 5.3: Example of Instances in Person Hierarchy

However, before selecting the views to materialize we need to take into account not only the explicit triples present in RDF but also the implicit ones. In RDF, these implicit triples are considered a part of the graph; they complete the query answer.

4.5 RDF Entailment

RDF entailment is a mechanism which derives the implicit triples based on a set of entailment rules. The W3C RDF Recommendation [2] defines a number

of patterns of entailment which lead to deriving implicit triples from RDF datasets. RDF Schema (RDFS) entailment patterns are particularly interesting since RDFS encodes the semantics of an application domain.

Accounting for implicit triples in the materialized views is important for returning a complete answer to the query since these triples are also considered a part of the RDF dataset. Materialized views process and group data and produce the aggregated summaries. These summaries of data do not contain the provenance information and cannot be used to extract the original triples. Thus, if the implicit triples are not accounted in materialized views, they will not be accounted when answering queries using views – the query answer will be incomplete.

It is important to note that the RDF entailment may derive new observations only based on existing information and specified rules. In this chapter we do not intend to propose a solution for adding missing knowledge to the database using logical rules. It is an orthogonal problem which is difficult to solve in general [84,85]. Neither do we intend to derive the information that is not known due to the Open World assumption. Instead, we focus on deriving missing information based on the existing data and specified semantics.

There are two main methods for processing queries when considering RDF entailment. In the *dataset saturation* approach, all implicit triples specified in the RDF recommendation are materialized and added to the dataset. Saturation allows to apply the plain query evaluation techniques on the saturated dataset to compute the answer. This technique also has some drawbacks – more space is needed for storing implicit triples and the maintenance is more complex since a change to a dataset requires recalculation of implicit triples. On the other hand, *query reformulation* leaves the dataset intact but reformulates a conjunctive query to a union of conjunctive queries. The answer can be obtained by executing this union of queries against the unchanged RDF data using standard query evaluation techniques. The drawback of this approach is the increasing overhead during query evaluation.

In MARVEL we use query reformulation techniques to produce and materialize the complete answer of a view query. By using query reformulation we leave a database intact but still account for implicit triples in query answers. Taking into account that the evaluation of the view query takes place once and the results are reused for other queries, we believe that this overhead is justified.

Table 5.1 lists some of the entailment rules used to derive facts from typical RDFS constraints. It specifies semantic relationships between classes and properties of a dataset. These rules can be used to reformulate a query into a union of conjunctive queries that will provide a complete answer over a non-saturated dataset. In our approach, we use the algorithm described in [79] for reformulating a query using RDFS rules.

RDFS Properties	Explicit Triple	Implicit Triple
(c_1, rdfs:subClassOf, c_2)	(s, rdf:type, c_1)	(s, rdf:type, c_2)
(p_1, rdfs:subPropertyOf, p_2)	(s, p_1, o)	(s, p_2, o)
(p, rdfs:range, c)	(s, p, o)	(o, rdf:type, c)
(p, rdfs:domain, c)	(s, p, o)	(s, rdf:type, c)

Table 5.1: RDFS Entailment Rules

4.6 MARVEL View Selection Algorithm

Given the open nature of SPARQL endpoints, we cannot assume that the set of queries is known in advance. Instead, we assume all groupings in user queries to be equally likely.

Suppose we want to materialize N views with the maximum benefit regardless of their size to use for answering user queries. Algorithm 4 presents our method for selecting the materialized views.

We start by assigning the cost to every view. At the beginning it is equal to the size of the dataset since the whole dataset is used to answer view defining queries. Then we compute the benefit of the candidate view for the cases when this candidate view is used to derive a full answer (single path roll-up) to another view in the cube lattice (lines 4-10). The benefit of the candidate view is computed according to the cost model defined in Section 4.4. Thus, we can find the benefits of all appropriate views of the cube lattice.

The same algorithm is applied when a view can only be fully computed from a set of views (multiple path roll-up – lines 11-17). In these cases all the views in the set are considered together.

After calculating the benefit of the views, the algorithm selects the view with the maximum benefit. If the number of views already selected for materialization together with the current view does not exceed N, the selected view is added to the set of the views proposed for the materialization. This is done to take into account that there can be a set of views for which the benefit is calculated together. After selecting a (set of) view(s) to materialize, the cost of the remaining views is recalculated to take into account the recently selected view(s) (lines 18-27). If the selected set of the views with maximum benefit does not conform to the initial conditions, the selected set is discarded from further considerations (line 27). This process is repeated until we identify all N views.

The complexity of this algorithm is $\mathcal{O}(n^2)$ due to the utilization of nested loops – the view benefits over other views are calculated for all views.

Input: Set of views W, cube schema S, number of needed views N
Output: Selected views $W^{'}$

$W^{'} = \emptyset$ -- set of selected views; $T = \emptyset$ -- set of discarded view sets;
$Cost(v_i) = Size(G)$ -- for all views where G is base data ;
while $|W^{'}| \neq N$ **do**
  $\{V, B\} = \emptyset$ -- set of views together with the benefit ;
  foreach *view* $w \in W$ **do**
    $B_w = 0$ -- benefit of the view w ;
    foreach *view* $v \in W$ **do**
      if $w \preceq v$ *(according to S) and roll-up is single path* **then**
        $B_w = B_w + (Cost(v) - Size(w))$;
    $\{V, B\} = \{V, B\} \cup (w, B_w)$;
  foreach $\{w_1...w_n\}$ *for which* $\exists v$ *such that* $v \preceq \{w_1...w_n\}$ *(according to S) and roll-up type is multiple path* **do**
    $B_{\{w_1...w_n\}} = 0$ -- benefit of the set of views $\{w_1...w_n\}$;
    foreach *view* $v \in W$ *where* $v \preceq \{w_1...w_n\}$ **do**
      if $(\sum_{i=1}^{n} Size(w_i)) < Cost(v)$ **then**
        $B_{\{w_1...w_n\}} = B_{\{w_1...w_n\}} + (Cost(v) - (\sum_{i=1}^{n} Size(w_i)))$;
    if $B_{\{w_1...w_n\}} \neq 0$ **then**
      $\{V, B\} = \{V, B\} \cup (\{w_1...w_n\}, B_{\{w_1...w_n\}})$;
  let w be the view (or set of views) from $\{V, B\}$ for which B_w (or $B_{\{w_1...w_n\}}$) is MAX ;
  if $|w| \leq (N - |W^{'}|)$ *and* $w \notin T$ **then**
    $W^{'} = W^{'} \cup w$; $\{V, B\} = \{V, B\} - (w, B_w)$;
    foreach *view* $v \in W$ **do**
      if $v \preceq w$ *(according to S)* **then**
        if $Size(w) < Cost(v)$ **then**
          $Cost(v) = Size(w)$;
    $W = W \setminus w$;
  else
    $T = T \cup w$
return $W^{'}$;

Algorithm 4: Algorithm for Selecting Views to Materialize

5 Query Rewriting in MARVEL

There are several aspects that complicate the problem of rewriting queries over SQL aggregate views. First, in SPARQL a user query and a view definition may use different variables to refer to the same entity. Thus, the query

rewriting algorithms require variable mapping to rewrite a query. A variable mapping maps elements of a triple pattern in the BGP of the view to the same elements of a triple pattern in the BGP of the query. Second, the algorithms need to match the new graph structure that is formed by the *CONSTRUCT* query of the view to the graph patterns of the user query and possibly aggregate and group these data anew. Third, complex and indirect hierarchies present in RDF data complicate query rewriting and need to be taken into consideration.

The rewriting algorithms proposed in [78, 79] target conjunctive queries and do not consider grouping and aggregation of data. Therefore, we built upon these algorithms and developed an algorithm to rewrite aggregate queries that identifies the views which may be used for query rewriting and selects the one with the least computational cost.

For ease of explanation, we split the algorithm used in MARVEL for aggregate query rewriting using views into two parts: an algorithm for identifying the best view for rewriting (Algorithm 5) and a query rewriting algorithm (Algorithm 6). In the algorithms, we need to look for dimension *roll-up paths* (RUPs), i.e., path-shaped joins of triple patterns of the form $\{(root, p_1, o_1), (s_2, p_2, o_2), ..., (s_n, p_n, d)\}$ where *root* is the root of the BGP, p_x is a predicate from the set of hierarchy steps defined for hierarchies in a cube schema, and triple patterns in the path are joined by subject and object values, e.g., $o_{x-1} = s_x$. We denote such a RUP as $\delta_{p_{dim}}(d_i)$ where p_{dim} is a predicate connecting the root variable to the first variable in the roll-up path and d_i represents the last variable in the path. These algorithms use $\gamma(agg_N)$ and $\gamma(g_N)$ to denote sets of triple patterns in the CONSTRUCT clause *CnPtrn* $\{(s, p^V_{C1}, g_1), \ldots, (s, p^V_{Cn}, g_n), (s, p^V_{Cm}, agg_m), \ldots, (s, p^V_{Ck}, agg_k)\}$ describing the results of aggregation, e.g., (s, p^V_{Cx}, agg_x), and grouping, e.g., (s, p^V_{Cx}, g_x).

The first step in Algorithm 5 is to replace all literals and IRIs in the user query with variables and corresponding *FILTER* statements (line 2): $(?s, p, \#o) \rightarrow (?s, p, ?o)$. $FILTER(?o = \#o)$. We do this to make graph patterns of views and queries more compatible with each other, since the graph patterns in the aggregated views should not contain literals. This may also potentially increase the number of candidate views since we may now use the views grouping by the hierarchy level of the replaced literal and then apply restrictions imposed by the *FILTER* statement.

To make the user query and the view query more compatible, we rename all variable names in the user query to the corresponding variable names in a view (line 4). We start from the root variable and replace all occurrences of this variable name in the user query with the name that is used in the view query. We then continue renaming variables that are directly connected to the previously renamed variables. We continue until we have renamed all corresponding variables in the user query.

Afterwards, for each dimension of the query graph pattern we define the

Input: Set of materialized views MV, query Q, data cube schema S
Output: Selected view w

1 $W = \emptyset$ -- Set of candidate views ;
2 $Q = ReplaceLiteralsAndURI(Q)$;
3 **foreach** $v \in MV$ **do**
4 $\quad Q = RenameVariables(Q, v)$;
5 $\quad \{d_1^Q, \dots d_n^Q\} = FindMinimalRUP(Q)$;
6 $\quad \{d_1^v, \dots d_n^v\} = FindMinimalRUP(v)$;
7 $\quad$ let $\{hlvl(d_1)^Q \dots hlvl(d_n)^Q\}$ be a set of hierarchy levels of Q defined in S ;
8 $\quad$ let $\{hlvl(d_1)^v \dots hlvl(d_m)^v\}$ be a set of hierarchy levels of v defined in S ;
9 $\quad agg^Q = \{\varphi(o_1), ..., \varphi(o_n)\}$ -- All aggregate expressions in Q ;
10 $\quad agg^v = \{\varphi(o_1), ..., \varphi(o_m)\}$ -- All aggregate expressions in v ;
11 $\quad$ **if** $agg^Q \subseteq agg^v$ *and* $(\{hlvl(d_1)^Q \dots hlvl(d_n)^Q\} \preceq \{hlvl(d_1)^v \dots hlvl(d_m)^v\})$ *such that* $hlvl(d_i)^Q \preceq hlvl(d_i)^v$ *for all* i **then**
12 $\quad\quad W = W \cup v$;
13 **return** $w \in W$ with minimal costs ;

Algorithm 5: Algorithm for selecting a candidate view

appropriate roll-up path that the candidate view should have (lines 5-6). This path depends on the conditions (*FILTER* statements) and/or grouping related to the corresponding hierarchy and is the minimum of both; we take the roll-up paths to variables in *FILTER* and *GROUP BY* for the same dimension and keep only the triple patterns that are the same in both – common RUP. For example, if the query groups by regions of a country but the *FILTER* statement restricts the returned values to only some cities ($Region \preceq City$), the required level of the hierarchy in the view should not be higher than the City level.

Then, we identify the hierarchy levels for all dimensions in the query and all dimensions in a view and compare them. We check that the hierarchy levels of all dimensions defined in the view do not exceed the needed hierarchy levels of the query and that the set of aggregate expressions defined in a view may be used to compute the aggregations defined in the query. The views complying with these conditions are added to the set of candidate views (line 12). Out of these views we select one with the least cost for answering the query (line 13).

Let us consider an example. Given the materialized view described in Listing 5.3 and the query of Listing 5.1, the system renames all variables in the query to the corresponding variable names in the view (i.e. *?place* $\rightarrow$ *?vCity*; *?fact* $\rightarrow$ *?obs*; *?val* $\rightarrow$ *?cons*) and defines the roll-up paths for the dimensions in the query (i.e. (*?fact gol:refBuilding/org:siteAddress/vc:adr/vc:locality ?vCity*) and (*?fact gol:reportDateTime ?date*)). Note that the roll-up path in the Date

dimension contains the Date level and not the Month level since the query groups by dates. Then the system identifies the roll-up paths for the dimensions in the view (i.e. (*?fact gol:refBuilding/org:siteAddress/vc:adr/vc:locality ?vCity*) and (*?fact gol:reportDateTime ?date*)) and compares them. The system also identifies aggregation expressions in the query and the view (*?fact qb:observation/gol:utilityConsumption ?cons, (SUM(?cons) as ?value)*). Since the view contains the same aggregate expression and all necessary dimensions and the hierarchy levels of the dimensions in the view do not exceed those in the query, this view is added to the set of candidate views.

Given one of the collected views, MARVEL uses Algorithm 6 to rewrite a query. For every dimension in the query we identify the common roll-up path in the query and the view. In the rewritten query Q', these triple patterns will be replaced by the triple patterns from the *CONSTRUCT* clause of the view ($\gamma^V(c^V)$). The remaining triple patterns belonging to the dimensions ($\Delta(d^Q)$) remain unchanged (lines 4-11).

Afterwards, the algorithm compares the aggregate functions of the query and the *SELECT* clause of the view and identifies those that are needed for rewriting. We add the corresponding triple pattern from the *CONSTRUCT* clause and rewrite the aggregate functions to account for the type of the function (algebraic or distributive) (lines 12-17). *GROUP BY* and *ORDER BY* clauses do not change. Additionally, the triple patterns of the *CONSTRUCT* clause will be placed inside the *GRAPH* statement of the SPARQL query to account for the different storage of the view triple patterns (lines 10, 16, 18).

```
SELECT ?date ?vCity (SUM(?value) as ?aggValue)
FROM <http://data.gov.uk>  FROM NAMED <http://data.gov.uk/matview1>
WHERE { GRAPH <http://data.gov.uk/matview1> { ?id gol:reportDate ?date;
    gol:reportLocality ?vCity;  gol:utilityConsumption ?value. }
  ?month skos:narrower ?date . ?month gol:value ?mVal .
  FILTER (?mVal = 'September 2015')  } GROUP BY ?date ?vCity
```

Listing 5.6: Rewritten query

Listing 5.6 shows the result of rewriting the query from Listing 5.1 using the view from Listing 5.3. The algorithm identifies common roll-up paths for the two dimensions in the view and in the query: *?fact gol:refBuilding/org:siteAddress/vc:adr/vc:locality ?vCity* and *?fact gol:reportDateTime ?date*. The system replaces these triple patterns with the triple pattern from the *CONSTRUCT* clause and puts these replaced triple patterns inside the *GRAPH* statement. The remaining triple patterns in the Date dimension (*?month skos:narrower ?date .* and *?month gol:value ?mVal .*) are added to the query graph pattern outside the *GRAPH* statement. The aggregate function is rewritten; since *SUM* is a distributive function, it is rewritten using the same aggregation (*SUM*). All assignment and constraint functions (e.g., *FILTER*) are copied to the rewritten query.

Input: View v, query Q
Output: Rewritten query Q'

$GP' = \emptyset$; $RD' = \emptyset$; $GBD = vars^{Q}_{GRP}$;
let Φ^{Q} be *assignment* and *constraint* functions of Q ;
$GBGP' = \emptyset$; -- A graph pattern of *GRAPH* statement ;
$qDims = \{\delta_p(d^{Q}) \dots\}$ -- Set of RUP in query Q ;
$vDims = \{\delta_p(d^{v}) \dots\}$ -- Set of RUP in view v ;
foreach $\delta_p(d^{Q}) \in qDims$ **do**
    $\delta_p(c^{Q}) = \delta_p(d^{Q}) \cap \delta_p(d^{v})$ -- Common RUP in Q and v ;
    $\Delta(d^{Q}) = \delta_p(d^{Q}) \setminus \delta_p(c^{Q})$ -- Remaining part of a RUP (remaining triple patterns) in Q after subtracting the part in common with v;
    let $\gamma^{v}(c^{v})$ be a triple pattern $\in CnPtrn$ such that $\gamma^{v}(c^{v})$ represents $\delta_p(c^{v})$;
    $GP' = GP' \cup \Delta(d^{Q})$; $GBGP' = GBGP' \cup \gamma^{v}(c^{v})$;
    $RD' = RD' \cup \{d^{Q}\}$;
$agg^{Q} = \{\varphi(o_1), ..., \varphi(o_n)\}$ -- Aggregate expressions in Q over variables $\{o_1 \dots o_n\}$;
$agg^{v} = \{\varphi(o_1), ..., \varphi(o_m)\}$ -- Aggregate expressions in v over variables $\{o_1 \dots o_m\}$;
foreach $\varphi^{Q}(x) \in agg^{Q}$ **do**
    let $\gamma^{v}(x)$ be a triple pattern $\in CnPtrn$ such that $\gamma^{v}(x)$ represents $\varphi^{v}(x) \in agg^{v}$ and $\varphi^{v} = \varphi^{Q}$;
    $GBGP' = GBGP' \cup \gamma^{v}(x)$;
    $RD' = RD' \cup \{f'(\gamma^{v}(x))\}$ where f' is a rewrite of the aggregate function φ ;
$GP' = GBGP' \cup GP' \cup \Phi^{Q}$);
Q' = *SELECT* RD' *WHERE* GP' *GROUP BY* GBD ;
return Q' ;

Algorithm 6: Algorithm for query rewriting using a view

6 Evaluation

To evaluate the performance gain for queries executed over materialized views against the queries over the raw data, we implemented MARVEL using the .NET Framework 4.0 and the dotNetRDF (http://dotnetrdf.org/) API with Virtuoso v07.10.3207 as triple store. The times reported in this section represent total response time, i.e., they include query rewriting and query execution. All queries were executed 5 times following a single warm-up run. The average runtime is reported for all queries. The triple store was installed on a machine running 64-bit Ubuntu 14.04 LTS with CPU Intel(R) Core(TM) i7-950, 24GB RAM, 600GB HDD.

Query Templates	Query#	Query Parameters for Various Selectivities
Template 1. Amount of revenue increase that would have resulted from eliminating certain company-wide discounts.	Q1	Discounts 1, 2, and 3 for quantities less than 25 shipped in 1993.
	Q2	Discounts 1, 2, and 3 for quantities less than 25 shipped in 01/1993.
	Q3	Discounts 5, 6, and 7 for quantities less than 35 shipped in week 6 of 1993.
Template 2. Revenue for some product classes, for suppliers in a certain region, grouped by more restrictive product classes and all years.	Q4	Revenue for 'MFGR_12' category, for suppliers in America
	Q5	Revenue for brands 'MFGR_2221' to 'MFGR_2228', for suppliers in Asia
	Q6	Revenue for brand 'MFGR_2239' for suppliers in Europe
Template 3. Revenue for some product classes, for suppliers in a certain region, grouped by more restrictive product classes and all years.	Q7	For Asian suppliers and customers in 1992-1997
	Q8	For US suppliers and customers in 1992-1997
	Q9	For specific UK cities suppliers and customers in 1992-1997
	Q10	For specific UK cities suppliers and customers in 12/1997
Template 4. Aggregate profit, measured by subtracting revenue from supply cost.	Q11	For American suppliers and customers for manufacturers 'MFGR_1' or 'MFGR_2'
	Q12	For American suppliers and customers for manufacturers 'MFGR_1' or 'MFGR_2' in 1997-1998
	Q13	For American customers and US suppliers for category 'MFGR_14' in 1997-1998

Table 5.2: SSB Queries

6.1 Datasets and Queries

Unfortunately, none of the benchmarks for SPARQL queries are applicable to our setup. All these benchmarks produce the complete set of data that should be present in the source – data generators do not have an option to withhold some data from being present in the source and instead generate the implicit data that can be used to derive the missing data. Moreover, the BSBM Business Intelligence Use Case benchmark that contains analytical queries [86] does not require reasoning to answer the queries correctly, has only a limited set of analytical queries and does not involve hierarchies and data cubes; and the LUBM benchmark [87] that is used to evaluate reasoning capabilities does not define analytical queries. Therefore, we decided to test our approach on 2 different datasets and adapt data generators and queries to our needs. All queries, schemas, and datasets are available at `http://extbi.cs.aau.dk/aggview`.

LUBM Dataset LUBM [87] features an ontology for the university domain; it creates synthetic OWL data scalable to an arbitrary size. The dataset describes universities, departments, students, professors, publication, courses, etc. We decided to build our data cube and corresponding queries on the information related to courses. In particular, we are interested in knowing the number of courses offered by departments, the type of the courses, the number of students taking courses, etc.

To introduce incompleteness in the data we changed the data generator so that approx. 30% of the information that relates staff to courses is missing.

Instead, we introduced information about the department that offers these courses (*lubm:offeringDepartment*). In such settings, counting the number of courses offered by departments becomes more challenging since the roll-up path Course → Staff → Department needs to be complemented by the roll-up path Course → Department and the aggregation of courses by Department cannot be answered by the results of the aggregation by Staff. A simplified schema of the data structure is presented in Figure 5.4. We generated 3 datasets containing 30, 100, and 300 universities. The number of generated triples in each dataset is given in Table 5.3.

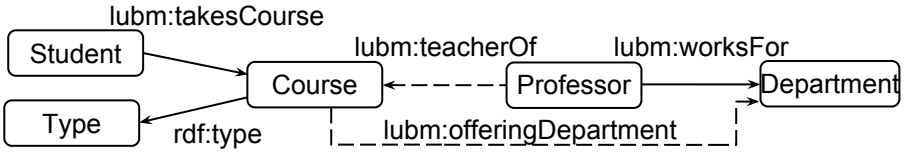


Fig. 5.4: Excerpt of an altered LUBM schema

# Universities	30	100	300
# Triples	3,968,866	13,405,383	39,874,037

Table 5.3: Materialized Views for LUBM Dataset

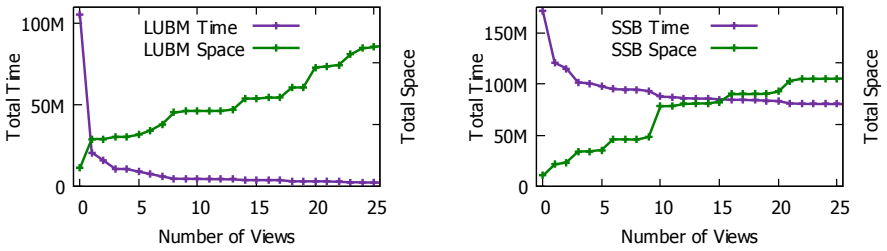
LUBM Queries Inspired by [80] and the corresponding technical report, we defined in SPARQL 6 analytical queries involving grouping over several classification dimensions. We use *COUNT* aggregation in all queries. These queries aggregate over number of courses offered by departments, number of courses taken by students, number of graduate courses in each department, number of courses taught by professors in each department, etc.
LUBM Cube Schema We drafted the QB4OLAP schema of the LUBM data cube specifying 3 dimensions (Student, Staff, and Course), hierarchy levels, and steps between the levels. In total, the schema contains 183 triples.
LUBM Materialized Views We applied Algorithm 4 to select a set of views providing a good performance gain for answering user queries. The execution of the algorithm on a data cube lattice with 60 nodes and known view sizes took 213 ms.

To choose which views to materialize, we ran MARVEL's view selection algorithm and measured (i) the total query response time for all queries in the cube using materialized views whenever possible and (ii) the total space these views require. The unit in which we measured both space and time consumption is the number of triples. The results for the first 25 views sorted by their benefit are presented in Figure 5.5a. Based on these results we decided

to materialize the first 5 views where the benefit in total response time for the views is good compared to the growth in space consumption for storing these views (Table 5.4).

View #	StudentDim	StaffDim	CourseDim
View 1	Student	Department	*All*
View 2	Student	Professor	*All*
View 3	Department	Professor	*All*
View 4	Department	Department	*All*
View 5	*All*	Department	Type

Table 5.4: Materialized Views for LUBM Dataset



(a) LUBM Dataset **(b)** SSB Dataset

Fig. 5.5: Time and space vs number of views

SSB RDF Dataset In our experiments we also used the Star Schema Benchmark (SSB) [66], originally designed for aggregate queries in relational database systems. This benchmark is well-known in the database community and was chosen for its well-defined testbed and its simple design.

The data in the SSB benchmark represent sales in a retail company; each transaction is defined as an observation described by 4 dimensions (Parts, Customers, Suppliers, and Dates). We translated the data into the RDF multidimensional representation (QB4OLAP) introducing incompleteness to this dataset as well, as illustrated in Figure 5.6. An observation is connected to dimensions (objects) via certain predicates. Every connected dimension object is in turn defined as a path-shaped subgraph. Hierarchies in dimensions are connected via the *skos:broader* predicate. Measures (represented as rectangles in Figure 5.6) are directly connected to observations. We changed the data generator to omit some information that relates suppliers to their corresponding cities in the Supplier dimension (and parts to their brands in the Part dimension). Instead, we connected suppliers with missing city information directly to their respective nations (*ssb:s_nation*) and parts with missing

Fig. 5.6: SSB Dataset in QB4OLAP format

brand information directly to the categories (*ssb:p_category*). Thus, in the roll-up path Supplier → City → Nation → Region the City level is incomplete. The Part dimension is affected in the level Brand (Part → Brand → Category → Manufacturer).

In our experiments, we used scaling factors 1 to 3 to obtain datasets of different sizes; the number of triples in each scaling factor are listed in Table 5.5. Observations and all dimensional data are stored in separate graphs – one for each dimension (parts, customers, suppliers, dates) and one for observations.

Scale Factor #	1	2	3
# Triples	122,327,740	244,518,460	364,744,560

Table 5.5: Number of Triples for the Generated SSB Test Datasets

SSB Queries SSB defines 13 queries. We converted all 13 defined queries into SPARQL. They are briefly described in Table 5.2.

SSB Materialized Views Then we applied Algorithm 4 to select a set of materialized views. The execution of the algorithm on a cube lattice with 500 nodes and known view sizes took 11.8 seconds. We then conducted the same time and space analysis as described above (Figure 5.5b). As a result, we

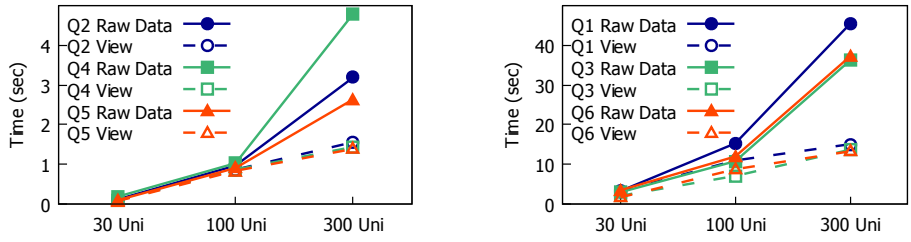
identified and materialized 6 views with the maximum benefit (Table 5.6) and stored these views in separate named graphs – each view in its own graph.

View #	Dates Level	Suppliers Level	Customers Level	Parts Level
View 1	Month	Supplier	City	Manufacturer
View 2	Month	Nation	Customer	Part
View 3	Year	Supplier	City	Category
View 4	Month	City	Nation	Part
View 5	Year	Nation	Region	Part
View 6	Date	*All*	Customer	Manufacturer

Table 5.6: Selected Materialized Views with Hierarchy Levels

6.2 Query Evaluation

LUBM Figure 5.7 shows the results of executing the LUBM queries for 3 scale factors – queries with similar runtimes are grouped into separate graphs for better visualization. For queries over raw data we materialized the implicit triples and saved them to the dataset to avoid the entailment during query execution. Note that the performance gain for queries over materialized views becomes more evident with the growth in the volume of data, due to the growing difference in their sizes. For scale factor 3 the execution of the queries over materialized views is on average 3 times faster.



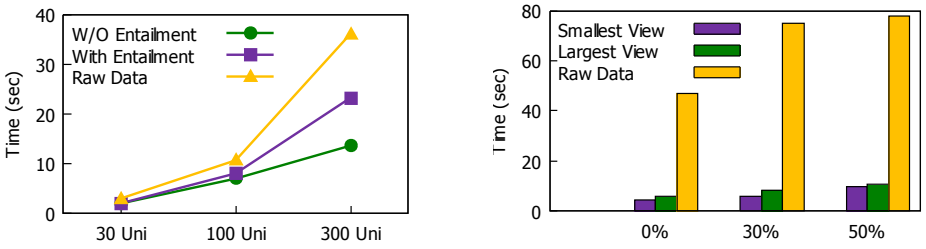
(a) Queries 2, 5 and 4

(b) Queries 1, 3 and 6

Fig. 5.7: Execution times of LUBM queries over raw data and views

We also compared the performance of the queries over views that take implicit triples into account and those that do not. Query 3 requests information on the number of courses taken by research assistants whose advisors are professors. We materialized 2 views: one takes into account that all professor ranks are subclasses of the more general class Professor and the other view does not. The execution of Query 3 over the view with implicit information

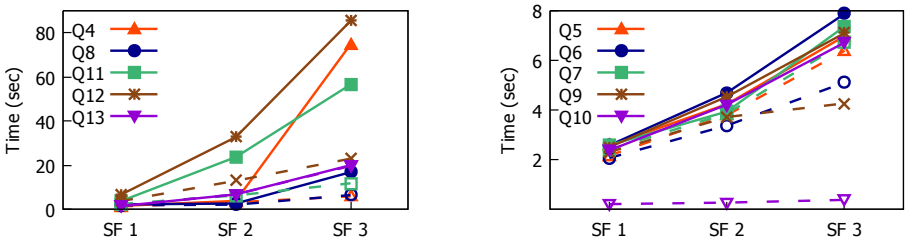
for scale factor 3 was 1.7 times faster than the execution over the other view (Figure 5.8a).



(a) Entailment in views, LUBM **(b)** Various incompleteness, SSB

Fig. 5.8: Execution times of LUBM and SSB queries

SSB Given the set of materialized views, MARVEL was able to rewrite 10 out of the 13 queries. The other 3 benchmark queries (Q1, Q2, and Q3) apply restrictions on measures. Since the views group by dimensions and only store aggregates over the measures, these queries cannot be evaluated on any aggregate view.



(a) SSB queries **(b)** SSB queries

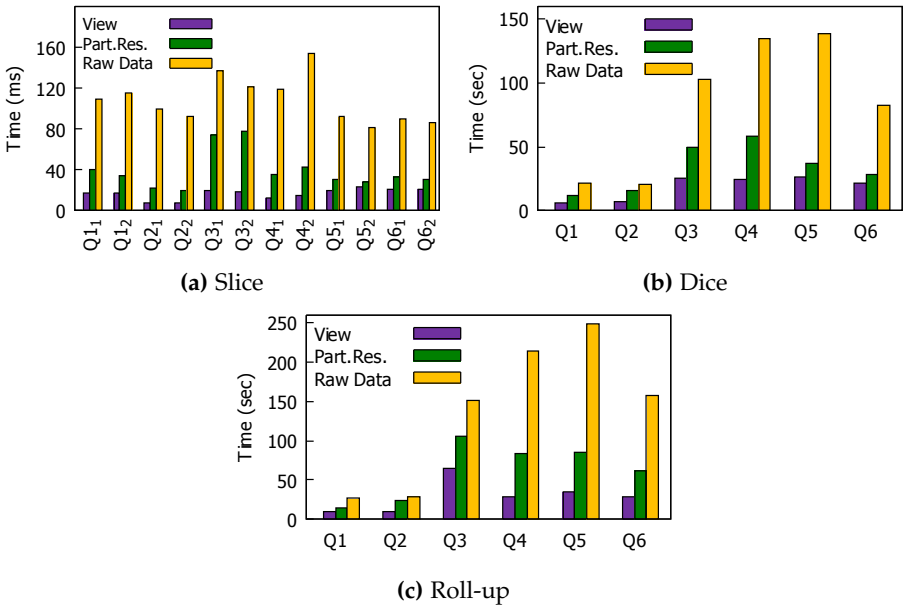
Fig. 5.9: Execution times of SSB queries over raw data and views

Figure 5.9 shows the runtime of the queries evaluated on the original datasets and on the views (dashed lines of the same colors indicate the execution times over views). Our results for scale factor 3 show that evaluating queries using views is on average 5 times faster (up to 18 times faster for Query 10). This can be explained by the decreased size of the data and the availability of partial results.

We also compared the performance gain for queries over views with different levels of incompleteness. For scale factor 3, we generated datasets with 0%, 30%, and 50% levels of incompleteness and identified a set of views for every dataset. In each case, the set of materialized views is different due to the difference in the size and the benefit of the views. We then evaluated the

execution of Query 4 over the raw data and over the largest and the smallest view. The slight increase in the query execution time over the raw data for incomplete datasets is caused by a rewriting of the query into a more complex query. The results show that in all cases the evaluation of queries over views is far more beneficial (on average 11 times more beneficial – Figure 5.8b).

Additionally, we compared the performance gain of MARVEL to the approach in [80] which materializes partial results of user queries to answer subsequent queries. We used the original (non-modified) LUBM dataset containing approx. 100M triples, analytical queries, and views introduced in the technical report of [80]. The execution times for the queries over the original data and views are reported in Figure 5.10. As shown in the figure, MARVEL is on average more than twice as fast as partial result materialization [80]. This can be explained by the difference in the size of the data – partial results contain identifiers for facts while our materialized views contain aggregated data only.



(a) Slice

(b) Dice

(c) Roll-up

Fig. 5.10: Comparison with results from [80]

In summary, the experimental results show that MARVEL accounts for RDF-specific requirements and finds an appropriate set of views that provide a good balance between the benefit of the views and their storage space. The rewriting algorithm of MARVEL is able to rewrite analytical SPARQL queries based on a set of materialized views. The experiments also show that evaluating queries over materialized views is on average 3-11 times faster

than evaluating the queries over raw data.

7 Conclusion and Future Work

In this chapter, we have addressed the problem of selecting a set of aggregate RDF views to materialize and proposed a cost model and techniques for choosing these views. The selected materialized views account for implicit triples present in the dataset. The chapter also proposes a SPARQL syntax for defining RDF views and an algorithm for rewriting user queries given a set of materialized RDF views. A comprehensive experimental evaluation showed the efficiency and scalability of MARVEL resulting in 3-10 times speedup in query execution. In future work, we plan to investigate algorithms for incrementally maintaining the materialized views in the presence of updates.

Chapter 6

Conclusion

Abstract

This chapter summarizes the conclusions and proposes directions for future work presented in Chapters 2-5.

1 Summary of Results

The capability of incorporating data from different sources into the decision making process is, nowadays, essential for BI applications. Huge amounts of data available on the Web call for more active use of these data to provide richer insights. However, due to the amount and complexity of data available on the Web, incorporation and utilization of these data pose challenges for technologies that were shaped for storing and manipulating structural data. Thus, this thesis presented our approach aimed at optimizing analytical queries over RDF sources. The thesis focused on data integration and data processing techniques to enable efficient execution of analytical SPARQL queries in the Semantic Web.

Overall, in the thesis we proposed a conceptual model of a system for Exploratory OLAP over RDF data sources. We have identified and described 4 modules needed in such systems. We also presented a use case to demonstrate the applicability of the proposed framework. We discussed the challenges and explored optimizations needed for the efficient processing of aggregate queries in a federation of SPARQL endpoints. In addition, we considered federated systems where endpoints contain related data divided by a certain "aspect" (like dimension, hierarchies, facts) to enable the analysis of data across these endpoints. At the end, we addressed the issue of improving the performance of SPARQL endpoints for aggregate queries using material-

ized RDF views. For each challenge that we addressed, we proposed algorithms and query processing optimization techniques that considered RDF specifics. In the sequel, we summarize contributions of each of the presented chapters.

Chapter 2 presented a framework for Exploratory OLAP over LOD sources. We followed the best practices in the system design and constructed the framework on a modular basis. This also helped us to investigate each module separately. The framework employs the schema of the data cube expressed by the combination of QB4OLAP and VoID vocabularies and stored in the Global Conceptual Schema Module. By combining the vocabularies and specifying how RDF data can be accessed using various protocols, we enabled the system to query remote data sources, extract and aggregate data, and build an OLAP cube. We also introduced a computer-aided process for discovering previously unknown data sources necessary for the given data cube and building a multidimensional schema. We presented a use case that demonstrated the applicability of the proposed framework. Overall, Chapter 2 motivated the research of this thesis and set the directions for further investigation. In particular, we envisioned/designed the Distributed Query Processing Module and in the following chapters we investigated the algorithms and techniques to increase the efficiency of executing analytical SPARQL queries in a distributed environment.

Chapter 3 presented our findings related to the problem of efficiently processing aggregate queries in a federation of SPARQL endpoints. While executing an aggregate query that retrieves information from a remote endpoint, the state-of-the-art triple stores timed out during the query execution since they used basic strategies. We examined this issue and proposed our solution. More particularly, we investigated several query processing strategies for this scenario such as Mediator Join, SemiJoin, and Partial Aggregation. Separately, these strategies do not contribute to the optimal solution for an ad-hoc query. Thus, we proposed a cost model that analyzes each query and chooses the best strategy in each case. Our CoDA approach for aggregate SPARQL queries estimates constants and result sizes for triple patterns, joins, grouping and aggregation present in the query and makes an informed decision on which strategy to apply. The comprehensive experimental evaluation, based on an RDF version of the widely used Star Schema Benchmark, showed that CoDA is efficient and scalable, able to pick the best query processing plan in different situations, and significantly outperforms current state-of-the art triple stores.

Chapter 4 continued our research on increasing the efficiency of evaluating analytical SPARQL queries in a federation of SPARQL endpoints. In this chapter we investigated federations of SPARQL endpoints where data are distributed in such a manner that *related* data are stored on multiple endpoints such that an endpoint contributes data for a certain "aspect" (dimen-

sions, hierarchies, facts, etc.) only. A typical example is the setup where each endpoint contains statistics data of a single country only. In our approach (LITE), we defined mappings to link a mediated (global) schema to source (local) schemas, extended RDF vocabularies to be used for mapping, and proposed a query rewriting algorithm that rewrites a globally defined query to queries for local endpoints. During the rewriting process, our algorithm also takes into account hierarchical information encoded in RDFS. Additionally, we use global heuristics to optimize rewritten queries and the previously implemented cost model to improve the federated queries that are executed on local endpoints. Our experimental evaluation showed the efficiency and scalability of the proposed approach. The advantage of LITE was even more evident for queries that retrieve hierarchical data from remote endpoints.

In Chapter 5, we investigated one of the methods to increase the performance of aggregate queries on a single SPARQL endpoint – materializing RDF views for their further utilization during query execution. MARVEL is a materialized view selection and analytical SPARQL query rewriting approach for RDF data. Its view selection algorithm is based on an RDF-specific cost model that proposes to materialize a set of views to be used during a query execution. While materializing views, the approach also accounts for implicit triples present in the view. MARVEL also proposes a SPARQL syntax for defining aggregate RDF views and constructing new triples from tabular data generated by SPARQL queries, and an algorithm for rewriting user queries based on a selected materialized RDF view. A comprehensive experimental evaluation showed the efficiency and scalability of MARVEL resulting in 3-10 times speedup in query execution.

In summary, we concentrated on designing a framework that facilitates analytical SPARQL queries over Semantic Web sources. In Chapter 2, we developed the framework that retrieves data from federated RDF sources while in Chapters 3-5 we investigated different aspects and proposed algorithms and query processing optimization techniques for improving query performance for analytical SPARQL queries in a federated setup. Our framework allows to virtually integrate several endpoints into a federated system and to perform analytical queries that retrieve data from remote endpoints. The performance of standalone endpoints in a federation can also be optimized using materialized views. Overall, when using our framework, we observe 7x times performance gain for analytical queries in a federation, and 3-11x times performance gain for analytical queries on single endpoints.

2 Future Directions

Several future directions exist for the work presented in this thesis. Thus, we next discuss these research directions.

The approach presented in Chapter 2 is our vision of the framework for Exploratory OLAP that queries distributed RDF sources, extracts and aggregates data, and presents the results to a user. Up to the moment, we have focused on the performance optimization techniques for aggregate SPARQL queries in stand-alone endpoints and in distributed settings. In Chapters 2 and 4, we also specified and extended the vocabularies used for defining the multidimensional schema of the OLAP cube and the mappings between the global schema of the system and local schemas of individual data sources. Thus, the next step in the extension of the framework is to investigate and automate the process of building global and local schemas and mappings between these schemas and to integrate the whole framework with current BI tools to improve the capabilities of BI systems. Besides, even though we addressed query performance issues in different chapters of the thesis, we believe that further optimization is possible and required. We point out possible improvements below.

Chapter 3 explores the execution of aggregate SPARQL queries in a federated setup. In our approach, we assumed that no information is known about standalone endpoints comprising the federation. However, the cost model can benefit from the information that may be available for some of the endpoints to more accurately predict the communication and processing costs. Additionally, the model may be enhanced to more accurately estimate the selectivity of predicates and *FILTER* statements in SPARQL queries by using the histograms that capture data distributions in federated endpoints. Moreover, more complex statistics with precomputed join result sizes and correlation information may be used to better estimate cardinalities. Also, the approach needs to be expanded to handle more complex queries e.g., with optional patterns or complex aggregation functions, those involving property paths and complex subqueries, etc. Another interesting direction for future work is investigating the influence of ontological constraints and inference/reasoning in the context of federated aggregate SPARQL queries.

We envision several improvements for our approach presented in Chapter 4. A further optimization of our work would be to develop a cost model for optimizing queries on a global level based on some statistical data in cases where such statistical data is available. Another direction for future work is developing methods and algorithms for automatically generating schemas for local and global sources and mappings between these schemas.

There are several improvements for the cost model presented in Chapter 5. The current cost model is formulated for a generic triple store. Thus, the model can be optimized if algorithms and/or auxiliary structures of a particular triple store are taken into the account. Additionally, the choice of the views to materialize can be refined based on the logs or the likely set of user queries. Moreover, algorithms for incrementally maintaining materialized views in the presence of updates need to be developed.

Overall, the thesis addressed the issues related to the efficient execution of analytical queries in a federation of SPARQL endpoints. Developing methods and algorithms for automatically devising analytical schemas and mappings for the framework is left as a future work. One of the interesting directions for future research is to consider entailment rules in federated settings, when the inferred data in one endpoint affects the results of query answering in others. Another direction for future is to address the problem of answering analytical federated queries using distributed materialized views. Separately, our developed techniques for answering SPARQL queries using materialized views and our optimization techniques for analytical queries in a federated setup speed up the performance of analytical SPARQL queries either in a federation or on standalone endpoints. Combining these two techniques, we can further optimize the performance of analytical queries over the Semantic Web sources. In addition, our work calls for a new line of research on the integration of data from heterogeneous (XML, CSV, XLS, Raster, etc.) data management systems.

References

[1] W3C. (2013) Data W3C. http://www.w3.org/standards/semanticweb/data. [Online]. Available: http://www.w3.org/standards/semanticweb/data

[2] W3C RDF Working Group. (2014) RDF - Semantic Web Standards. http://www.w3.org/standards/semanticweb/data. [Online]. Available: https://www.w3.org/RDF/

[3] W3C. (2008) SPARQL Query Language for RDF. https://www.w3.org/TR/rdf-sparql-query/. [Online]. Available: https://www.w3.org/TR/rdf-sparql-query/

[4] ——. (2013) SPARQL 1.1 Federated Query. https://www.w3.org/TR/sparql11-federated-query/. [Online]. Available: https://www.w3.org/TR/sparql11-federated-query/

[5] A. Abelló, O. Romero, T. B. Pedersen, R. Berlanga, V. Nebot, M. J. Aramburu, and A. Simitsis, "Using semantic web technologies for exploratory OLAP: A survey," *TKDE*, vol. 99, 2014.

[6] W3C RDF Working Group. (2014) RDF Schema 1.1. https://www.w3.org/TR/rdf-schema. [Online]. Available: https://www.w3.org/TR/rdf-schema

[7] W3C. (2013) SPARQL 1.1 Query Language. https://www.w3.org/TR/sparql11-query/. [Online]. Available: https://www.w3.org/TR/sparql11-query/

[8] T. Berners-Lee, "Linked Data. W3C Design Issues," http://www.w3.org/DesignIssues/LinkedData.html, 2006.

[9] L. Etcheverry and A. A. Vaisman, "QB4OLAP: A vocabulary for OLAP cubes on the semantic web," in *COLD*, 2012.

[10] W3C. (2010) Describing linked datasets with the VoID vocabulary. http://www.w3.org/TR/void/. [Online]. Available: http://www.w3.org/TR/void/

[11] A. Sheth and J. Larson, "Federated database systems for managing distributed, heterogeneous, and autonomous databases," *ACM Computing Surveys*, vol. 22, no. 3, pp. 183–236, 1990.

[12] P. Heim, S. Hellmann, J. Lehmann, S. Lohmann, and T. Stegemann, "RelFinder: Revealing relationships in RDF knowledge bases," in *SAMT*, 2009, pp. 182–187.

[13] H. Inoue, T. Amagasa, and H. Kitagawa, "An ETL framework for online analytical processing of linked open data," in *WAIM*, 2013, pp. 111–117.

[14] T. Neumann and G. Moerkotte, "Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins," in *ICDE*, 2011, pp. 984–994.

[15] E. Oren, R. Delbru, M. Catasta, R. Cyganiak, H. Stenzhorn, and G. Tummarello, "Sindice.com: a document-oriented lookup index for open linked data," *IJMSO*, vol. 3, no. 1, pp. 37–52, 2008.

[16] O. Romero and A. Abelló, "Automating multidimensional design from ontologies," in *DOLAP*. ACM, 2007, pp. 1–8.

[17] O. Görlitz and S. Staab, "Federated data management and query optimization for linked open data," in *New Directions in Web Data Management*. Springer Berlin Heidelberg, 2011, pp. 109–137. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17551-0_5

[18] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt, "FedX: Optimization techniques for federated query processing on linked data," in *ISWC*, 2011, pp. 601–616.

[19] C. Buil-Aranda, A. Hogan, J. Umbrich, and P.-Y. Vandenbussche, "SPARQL web-querying infrastructure: Ready for action?" in *ISWC*, 2013, pp. 277–293. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41338-4_18

[20] TopQuadrant, "SPIN - SPARQL Syntax (W3C Member Submission 22 Feb 2011)," https://www.w3.org/Submission/spin-sparql/.

[21] W3C. (2013) W3C Semantic Web Activity Homepage. http://www.w3.org/2001/sw. [Online]. Available: http://www.w3.org/2001/sw/

[22] A. Abelló, J. Darmont, L. Etcheverry, M. Golfarelli, J. Mazón, F. Naumann, T. B. Pedersen, S. Rizzi, J. Trujillo, P. Vassiliadis, and G. Vossen, "Fusion cubes: Towards self-service business intelligence," *IJDWM*, vol. 9, no. 2, pp. 66–88, 2013.

[23] S. Web. (2013) SPARQL endpoint. http://semanticweb.org/wiki/SPARQL_endpoint. [Online]. Available: http://semanticweb.org/wiki/SPARQL_endpoint

[24] A. Vaisman and E. Zimányi, *Data Warehouse Systems: Design and Implementation*. Springer, 2014.

[25] L. Etcheverry, A. Vaisman, and E. Zimányi, "Modeling and querying data warehouses on the semantic web using QB4OLAP," in *DaWaK*, 2014, pp. 45–56.

[26] B. Kämpgen and A. Harth, "No size fits all - running the star schema benchmark with SPARQL and RDF aggregate views," in *ESWC*, 2013, pp. 290–304.

[27] D. Pedersen, K. Riis, and T. B. Pedersen, "XML-extended OLAP querying," in *SSDBM*, 2002, pp. 195–206.

[28] V. Nebot and R. Berlanga, "Building data warehouses with semantic web data," *Decision Support Systems*, vol. 52, no. 4, pp. 853–868, 2012.

[29] B. Kämpgen and A. Harth, "Transforming statistical linked data for use in OLAP systems," in *I-SEMANTICS*, 2011, pp. 33–40.

[30] W3C. (2013) The RDF data cube vocabulary. http://www.w3.org/TR/2013/CR-vocab-data-cube-20130625/. [Online]. Available: http://www.w3.org/TR/2013/CR-vocab-data-cube-20130625/

[31] B. Kämpgen, S. O'Riain, and A. Harth, "Interacting with statistical linked data via OLAP operations," in *ILD*, 2012, pp. 336–353.

[32] O. Hartig, "Zero-knowledge query planning for an iterator implementation of link traversal based query execution," in *ESWC*, 2011, pp. 154–169.

[33] C. Pedrinaci, D. Liu, M. Maleshkova, D. Lambert, J. Kopecky, and J. Domingue, "iServe: a linked services publishing platform," in *ORES*, 2010.

[34] C. Pedrinaci and J. Domingue, "Toward the next wave of services: Linked services for the web of data." *J.UCS*, vol. 16, no. 13, pp. 1694–1719, 2010.

[35] U. Bojars, A. Passant, F. Giasson, and J. G. Breslin, "An architecture to discover and query decentralized RDF data," in *SFSW*, 2007.

[36] A. Hogan, A. Harth, J. Umbrich, S. Kinsella, A. Polleres, and S. Decker, "Searching and browsing linked data with SWSE: the semantic web search engine," *J. Web Sem.*, vol. 9, no. 4, pp. 365–401, 2011.

[37] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K. Sattler, and J. Umbrich, "Data summaries for on-demand queries over linked data," in *WWW*, 2010, pp. 411–420.

[38] J. Umbrich, K. Hose, M. Karnstedt, A. Harth, and A. Polleres, "Comparing data summaries for processing live queries over linked data," *WWWJ*, vol. 14, no. 5-6, pp. 495–544, 2011.

[39] F. Prasser, A. Kemper, and K. Kuhn, "Efficient distributed query processing for autonomous RDF databases," in *EDBT*, 2012, pp. 372–383.

[40] O. Görlitz and S. Staab, "SPLENDID: SPARQL endpoint federation exploiting VoID descriptions," in *COLD*, 2011.

[41] S. Hagedorn, K. Hose, K.-U. Sattler, and J. Umbrich, "Resource Planning for SPARQL Query Execution on Data Sharing Platforms," in *COLD*, 2014.

[42] G. Ladwig and T. Tran, "Linked data query processing strategies," in *ISWC*, 2010, pp. 453–469.

[43] J. Umbrich, M. Karnstedt, A. Hogan, and J. X. Parreira, "Hybrid SPARQL queries: Fresh vs. fast results," in *ISWC*, 2012, pp. 608–624.

[44] K. Hose and R. Schenkel, "Towards benefit-based RDF source selection for SPARQL queries," in *SWIM*, 2012, pp. 2:1–2:86.

[45] T. Berners-Lee, J. Hendler, O. Lassila *et al.*, "The semantic web," *Scientific American*, vol. 284, no. 5, pp. 28–37, 2001.

[46] N. Shadbolt, W. Hall, and T. Berners-Lee, "The semantic web revisited," *IEEE Intelligent Systems*, vol. 21, no. 3, pp. 96–101, 2006.

[47] W3C. (2013) SPARQL 1.1 Overview. https://www.w3.org/TR/sparql11-overview/. [Online]. Available: https://www.w3.org/TR/sparql11-overview/

[48] M. Wick, "Geonames geographical database," http://www.geonames.org.

[49] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives, "DBpedia: A nucleus for a Web of open data," in *ISWC*, 2007.

[50] C. Buil-Aranda, M. Arenas, O. Corcho, and A. Polleres, "Federating queries in SPARQL 1.1: Syntax, semantics and evaluation," *J. Web Sem.*, vol. 18, no. 1, pp. 1–17, 2013.

[51] M. Arenas, C. Gutierrez, D. P. Miranker, J. Pérez, and J. F. Sequeda, "Querying Semantic Data on the Web?" *SIGMOD Rec.*, vol. 41, no. 4, pp. 6–17, 2013.

[52] C. Buil-Aranda, A. Polleres, and J. Umbrich, "Strategies for executing federated queries in SPARQL 1.1," in *ISWC*, 2014, pp. 390–405. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11915-1_25

[53] D. Kontokostas, P. Westphal, S. Auer, S. Hellmann, J. Lehmann, R. Cornelissen, and A. Zaveri, "Test-driven Evaluation of Linked Data Quality," in *WWW*, 2014, pp. 747–758.

[54] World Wide Web Consortium, "Describing Linked Datasets with the VoID Vocabulary (W3C Interest Group Note 03 March 2011)," http://www.w3.org/TR/void/.

[55] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus, "ANAPSID: An adaptive query processing engine for SPARQL endpoints," in *ISWC*, 2011, pp. 18–34. [Online]. Available: https://doi.org/10.1007/978-3-642-25073-6_2

[56] A. Deshpande, Z. Ives, and V. Raman, "Adaptive query processing," *Foundations and Trends in Databases*, vol. 1, no. 1, pp. 1–140, 2007.

[57] T. Urhan and M. J. Franklin, "XJoin: A reactively-scheduled pipelined join operator," *IEEE Data Eng. Bull.*, vol. 23, no. 2, pp. 27–33, 2000.

[58] G. Ladwig and T. Tran, "SIHJoin: Querying Remote and Local Linked Data," in *ESWC*, 2011, pp. 139–153. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21034-1_10

[59] C. Basca and A. Bernstein, "Avalanche: Putting the Spirit of the Web back into Semantic Web Querying," in *SSWS*, 2010.

[60] Z. Akar, T. G. Halaç, E. E. Ekinci, and O. Dikenelli, "Querying the web of interlinked datasets using VoID descriptions," in *LDOW*, 2012.

[61] M. Wylot, J. Pont, M. Wisniewski, and P. Cudré-Mauroux, "dipLODocus[RDF] - short and long-tail RDF analytics for massive webs of data," in *ISWC*, 2011, pp. 778–793. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-25073-6_49

[62] C. Buil-Aranda, M. Arenas, and Ó. Corcho, "Semantics and optimization of the SPARQL 1.1 federation extension," in *ESWC*, 2011, pp. 1–15.

[63] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total," in *ICDE*, 1996, pp. 152–159. [Online]. Available: http://dx.doi.org/10.1109/ICDE.1996.492099

[64] K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao, "Describing linked datasets," in *LDOW*, 2009. [Online]. Available: http://ceur-ws.org/Vol-538/ldow2009_paper20.pdf

[65] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database systems - the complete book*, 2nd ed. Pearson Education, 2009.

[66] P. O'Neil, E. J. O'Neil, and X. Chen, "The star schema benchmark (SSB)," http://www.cs.umb.edu/~poneil/StarSchemaB.pdf, UMass/-Boston, Tech. Rep., 2009.

[67] T. P. P. Council, "TPC benchmark H – decision support," http://www.tpc.org/tpch.

[68] D. Colazzo, F. Goasdoué, I. Manolescu, and A. Roatis, "RDF analytics: lenses over semantic graphs," in *WWW*, 2014, pp. 467–478.

[69] D. Ibragimov, K. Hose, T. B. Pedersen, and E. Zimányi, "Processing aggregate queries in a federation of SPARQL endpoints," in *ESWC*, 2015, pp. 269–285.

[70] ——, "Towards exploratory OLAP over linked open data - A case study," in *BIRTE*, 2014, pp. 114–132.

[71] A. Halevy, "Answering queries using views: A survey," *VLDB Journal*, vol. 10, no. 4, pp. 270–294, 2001.

[72] D. Srivastava, S. Dar, H. Jagadish, and A. Levy, "Answering Queries with Aggregation Using Views," in *VLDB*, 1996, pp. 318–329.

[73] A. Gupta, V. Harinarayan, and D. Quass, "Aggregate-Query Processing in Data Warehousing Environments," in *VLDB*, 1995, pp. 358–369.

[74] V. Harinarayan, A. Rajaraman, and J. Ullman, "Implementing data cubes efficiently," in *ACM SIGMOD*, vol. 25, no. 2, 1996, pp. 205–216.

[75] D. Theodoratos, S. Ligoudistianos, and T. Sellis, "View selection for designing the global data warehouse," *DKE*, vol. 39, no. 3, pp. 219–240, 2001.

[76] S. Agrawal, S. Chaudhuri, and V. Narasayya, "Automated selection of materialized views and indexes in SQL databases," in *VLDB*, 2000, pp. 496–505.

[77] R. Castillo, C. Rothe, and U. Leser, "RDFMatView: Indexing RDF Data using Materialized SPARQL queries," in *SSWS*, 2010.

[78] W. Le, S. Duan, A. Kementsietsidis, F. Li, and M. Wang, "Rewriting queries on SPARQL views," in *WWW*, 2011, pp. 655–664.

[79] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu, "View selection in semantic web databases," *PVLDB*, vol. 5, no. 2, pp. 97–108, 2011.

[80] E. A. Azirani, F. Goasdoué, I. Manolescu, and A. Roatis, "Efficient OLAP operations for RDF analytics," in *ICDE Workshops*, 2015, pp. 71–76.

[81] C. Buil-Aranda, M. Arenas, Ó. Corcho, and A. Polleres, "Federating queries in SPARQL 1.1: Syntax, semantics and evaluation," *Web Semantics*, vol. 18, no. 1, pp. 1–17, 2013.

[82] A. Shukla, P. Deshpande, J. F. Naughton, and K. Ramasamy, "Storage estimation for multidimensional aggregates in the presence of hierarchies," in *VLDB*, 1996, pp. 522–531.

[83] F. Darari, W. Nutt, G. Pirrò, and S. Razniewski, "Completeness statements about RDF data sources and their use for query answering," in *ISWC*, 2013, pp. 66–83.

[84] L. A. Galárraga, C. Teflioudi, K. Hose, and F. M. Suchanek, "AMIE: association rule mining under incomplete evidence in ontological knowledge bases," in *WWW*, 2013, pp. 413–422.

[85] Z. Abedjan and F. Naumann, "Amending RDF entities with new facts," in *Know@LOD*, 2014.

[86] C. Bizer and A. Schultz, "The berlin SPARQL benchmark," *Int. J. Semantic Web Inf. Syst.*, vol. 5, no. 2, pp. 1–24, 2009.

[87] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *J. Web Sem.*, vol. 3, no. 2-3, pp. 158–182, 2005.

[88] X. Yin and T. B. Pedersen, "Evaluating XML-extended OLAP queries based on a physical algebra," in *DOLAP*, 2004, pp. 73–82.

[89] A. Doan, A. Halevy, and Z. Ives, *Principles of Data Integration*. Morgan Kaufmann, 2012.

[90] D. Pedersen, K. Riis, and T. B. Pedersen, "Query optimization for OLAP-XML federations," in *DOLAP*, 2002, pp. 57–64.

[91] T. B. Pedersen, D. Pedersen, and J. Pedersen, "Integrating XML data in the TARGIT OLAP system," *Int. J. Web Eng. Technol.*, vol. 4, no. 4, pp. 495–533, 2008.

[92] G. Correndo, M. Salvadores, I. Millard, H. Glaser, and N. Shadbolt, "SPARQL query rewriting for implementing data integration over linked data," in *EDBT/ICDT Workshops*, 2010.

[93] K. Makris, N. Bikakis, N. Gioldasis, and S. Christodoulakis, "SPARQL-RW: transparent query access over mapped RDF data sources," in *EDBT*, 2012, pp. 610–613.

[94] G. Montoya, L. D. Ibáñez, H. Skaf-Molli, P. Molli, and M. Vidal, "Semlav: Local-as-view mediation for SPARQL queries," *T-LSD-KCS*, vol. 13, pp. 33–58, 2014.

[95] D. Ibragimov, K. Hose, T. B. Pedersen, and E. Zimányi, "Optimizing aggregate SPARQL queries using materialized RDF views," in *ISWC*, 2016, pp. 341–359.

[96] G. Montoya, H. Skaf-Molli, P. Molli, and M. Vidal, "Federated SPARQL queries processing with replicated fragments," in *ISWC*, 2015, pp. 36–51.

[97] B. Quilitz and U. Leser, "Querying distributed RDF data sources with SPARQL," in *ESWC*, 2008, pp. 524–538.

[98] M. Saleem and A. N. Ngomo, "HiBISCuS: Hypergraph-based source selection for SPARQL endpoint federation," in *ESWC*, 2014, pp. 176–191.

[99] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and Complexity of SPARQL," in *ISWC*, 2006, pp. 30–43.

[100] T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*, 3rd ed. Springer, 2011.

ISSN (online): 2446-1628
ISBN (online): 978-87-7210-072-2

AALBORG UNIVERSITY PRESS